云栖社区 系列丛书

HZ BOOKS 华章IT

RocketMQ

实战与原理解析

杨开元◎著

阿里巴巴数据专家/RocketMQ源码贡献者撰写，RocketMQ官方开发团队鼎力推荐

APACHE ROCKETMQ

PRINCIPLE AND PRACTICE

机械工业出版社
China Machine Press

云栖社区系列

RocketMQ实战与原理解析

杨开元　著

ISBN：978-7-111-60025-1

本书纸版由机械工业出版社于2018年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

推荐序

在阿里巴巴技术发展初期，伴随着淘宝业务的快速发展，网站流量呈现几何级增长。单体巨无霸式的应用无法处理爆发式增长的流量，阿里内部从业务、组织层面进行了一次大的水平与垂直切分，拆分出用户中心、商品中心、交易中心、评价中心等平台型应用，分布式电商系统的雏形由此诞生。阿里的消息引擎就是在这样的大背景下诞生的，并被应用于各个应用系统之间的异步解耦和削峰填谷。

从最初的日志传输领域到后来阿里集团全维度在线业务的支撑，RocketMQ被广泛用于交易、数据同步、缓存同步、IM通讯、流计算、IoT等场景。在近几年的双11全球狂欢节中，RocketMQ以万亿级的消息总量支撑了全集团3000多个应用，为复杂的业务场景提供了系统解耦、削峰填谷的能力，保障了核心交易链路消息流转的低延迟、高吞吐，为阿里集团大中台的稳定性发挥了举足轻重的作用。

为了更好地发展RocketMQ社区生态，2016年双11前后，阿里巴巴将RocketMQ捐赠给Apache基金会，吸引了全球的开源爱好者参与到RocketMQ社区中，并于2017年9月成为Apache基金会的顶级项目。在开源社区的帮助下，RocketMQ具备了对接主流大数据流计算平台、离在线数据处理以及对接存储平台的能力。

本书介绍了分布式消息中间件RocketMQ的方方面面，作者为大数据领域的技术专家，在分布式领域具有很丰富的理论积累和实战经验。书如其人，书中各章节尽展实战经验，庖丁解牛般剖析了Apache RocketMQ的原理和架构设计。本书深入浅出地分析了RocketMQ的整体架构，分享了部署和运维的经验，涵盖RocketMQ的核心特性——高可用、高可靠机制，以及开源生态等。

本书作为国内首本全面解析Apache RocketMQ的书籍，对于希望了解RocketMQ技术内幕，以及想要掌握分布式系统设计理念的技术人员来说的确不容错过。

——周新宇，Apache RocketMQ项目管理委员会成员

前言

为什么要写这本书

几年前在做一个项目的时候，若需要用到消息队列，简单调研一下就会决定用Kafka，因为当时还不知道有RocketMQ。在我加入阿里后，当时有个项目需要用到消息中间件，试用了RocketMQ，发现阿里开源的消息中间件性能非常强大，但是上手有点费劲，因为现有文档多是零零散散的博文。在没有合适文档指导的情况下，对系统中用到的RocketMQ模块心里没底，系统偶尔出现异常时总会束手无策，需要通过查看很多源码，才能保证系统的稳定运行。

熟悉RocketMQ以后，我发现它是一款非常优秀的中间件产品，可以确保不丢消息，而且效率很高。同时因为它是用Java开发的，所以修改起来比较容易。

在阿里内部，RocketMQ很好地服务了集团大大小小上千个应用，在每年的双十一当天，更有不可思议的万亿级消息通过RocketMQ流转（在2017年的双11当天，整个阿里巴巴集团通过RocketMQ流转的线上消息达到了万亿级，峰值TPS达到5600万），在阿里大中台策略上发挥着举足轻重的作用。所以如果有合适的参考文档，RocketMQ会被更多人接受和使用，让更多人不必重复造“轮子”。

我做了很多年开发，在学校课本上学的开发知识有限，大多数是通过看书和上网学到的，其中很多优秀的文章对自己帮助很大。所以我很希望能用这本书回馈技术社区中有需要的开发者们。

动笔写这本书前，我系统地阅读了RocketMQ的源码，有些理解不够透彻的地方请教了阿里RocketMQ开发团队的同事，然后也总结了自己多年实际工作中的一些经验。希望这本书能简明扼要地说清楚RocketMQ的使用方法和核心原理。

读者对象

·希望学习分布式系统或分布式消息队列的开发人员。

·服务端系统开发者，他们可以借助高质量中间件来提高开发效率。

·软件架构师，他们可以通过消息队列优化复杂系统的设计。

本书特色

本书系统地介绍了RocketMQ这款优秀的分布式消息队列软件，通过阅读本书，读者可以快速把RocketMQ应用到自己的项目中，也可以通过更改源码定制符合自身业务的消息中间件。

如何阅读本书

本书分为两大部分：

第一部分是RocketMQ实战，包括第1～8章。这是本书的主体内容，可帮助读者快速用好RocketMQ这个分布式消息队列。

这部分是按照由浅入深的方式撰写的，为了让读者快速上手，首先介绍了搭建一个简单RocketMQ集群的方法，以此来发送和接收消息；然后详细介绍了如何用好Consumer和Producer，如何选择合适的类以及进行参数设置；再进一步根据应用，说明如何让RocketMQ在各种异常情况下保持稳定可靠，以及如何增大RocketMQ的吞吐量，从而在单位时间内处理更多的消息。

第二部分是源码分析，包括第9～13章。当读者有特殊的业务需求，需要更改或扩展RocketMQ现有功能的时候，这部分内容能帮助读者快速熟悉源码，找到要下手更改的地方，快速实现想要的功能。

这部分也适合想通过源码，深入学习消息队列的读者阅读。学习别人优秀的代码是提升自己技术水平的一条有效途径。

勘误和支持

由于水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。有任何的意见或建议，都可以通过邮箱rocketmqqa@163.com和我联系，真挚期待你的反馈。

致谢

写技术书籍很耗费时间，加之互联网行业快节奏的工作方式，导致我写这本书的时间大多是在周末和夜晚。在此感谢家人对我的支持和理解，尤其感谢我的妻子，没有她对家庭的照顾和对我的鼓励，这本书是无法完成的。

感谢阿里消息中间件团队的Leader王小瑞，是你从技术和写作思路上给我很大的帮助。感谢消息中间件团队的其他同学，你们为开源社区贡献了一个高质量的软件，你们写的很多高质量博文使开发者更容易理解RocketMQ。

感谢机械工业出版社的编辑杨福川、张锡鹏，感谢云栖社区的刁云怡，阿里的校友耿嘉安，是你们始终支持我的写作，你们的引导和帮助使我能顺利完成全部书稿。

谨以本书献给我最亲爱的家人，以及众多热爱软件开发工作的朋友们！

杨开元

第1章　快速入门

本章可以让读者了解RocketMQ和分布式消息队列的功能，然后搭建好单机版的消息队列，进而能够发送并接收简单的消息。

1.1 消息队列功能介绍

简单来说，消息队列就是基础数据结构课程里“先进先出”的一种数据结构，但是如果要消除单点故障，保证消息传输的可靠性，并且还能应对大流量的冲击，对消息队列的要求就很高了。现在互联网“微架构”模式兴起，原有大型集中式的IT服务因为各种弊端，通常被分拆成细粒度的多个“微服务”，这些微服务可以在一个局域网内，也可能跨机房部署。一方面对服务之间松耦合的要求越来越高，另一方面，服务之间的联系却越来越紧密，对通信质量的要求也越来越高。分布式消息队列可以提供应用解耦、流量消峰、消息分发等功能，已经成为大型互联网服务架构里标配的中间件。

1.1.1 应用解耦

复杂的应用里会存在多个子系统，比如在电商应用中有订单系统、库存系统、物流系统、支付系统等。这个时候如果各个子系统之间的耦合性太高，整体系统的可用性就会大幅降低。多个低错误率的子系统强耦合在一起，得到的是一个高错误率的整体系统。

以电商应用为例，用户创建订单后，如果耦合调用库存系统、物流系统、支付系统，任何一个子系统出了故障或者因为升级等原因暂时不可用，都会造成下单操作异常，影响用户使用体验。

如图1-1所示，当转变成基于消息队列的方式后，系统可用性就高多了，比如物流系统因为发生故障，需要几分钟的时间来修复，在这几分钟的时间里，物流系统要处理的内容被缓存在消息队列里，用户的下单操作可以正常完成。当物流系统恢复后，补充处理存储在消息队列里的订单信息即可，终端用户感知不到物流系统发生过几分钟的故障。

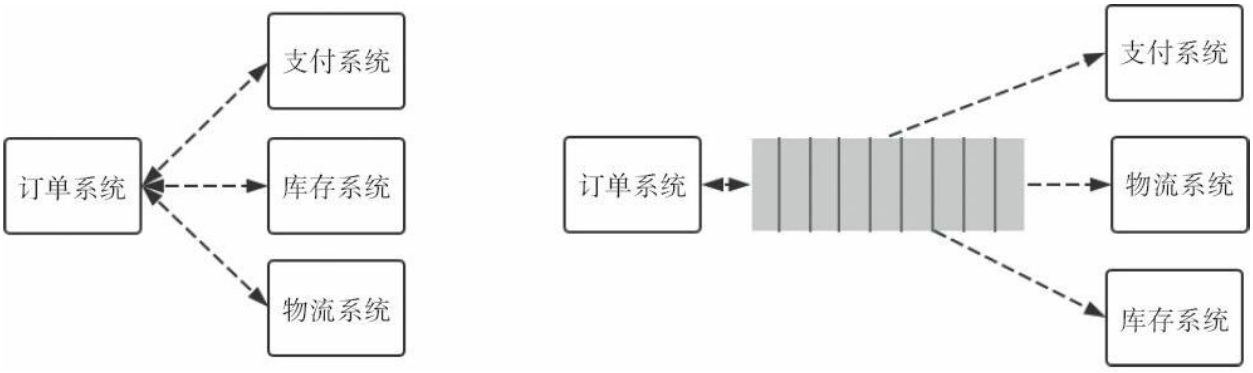


图1-1　消息队列的解耦功能

1.1.2 流量消峰

每年的双十一，淘宝的很多活动都在0点的时候开启，大部分应用系统流量会在瞬间猛增，这个时候如果没有缓冲机制，不可能承受住短时大流量的冲击。通过利用消息队列，把大量的请求暂存起来，分散到相对长的一段时间内处理，能大大提高系统的稳定性和用户体验。

举个例子，如果订单系统每秒最多能处理一万次下单，这个处理能力应对正常时段的下单是绰绰有余的，正常时段我们下单后一秒内就能返回结果。在双十一零点的时候，如果没有消息队列这种缓冲机制，为了保证系统稳定，只能在订单超过一万次后就不允许用户下单了；如果有消息队列做缓冲，我们可以取消这个限制，把一秒内下的订单分散成一段时间来处理，这时有些用户可能在下单后十几秒才能收到下单成功的状态，但是也比不能下单的体验要好。

使用消息队列进行流量消峰，很多时候不是因为能力不够，而是出于经济性的考量。比如有的业务系统，流量最高峰也不会超过一万QPS，而平时只有一千左右的QPS。这种情况下我们就可以用个普通性能的服务器（只支持一千左右的QPS就可以），然后加个消息队列作为高峰期的缓冲，无须花大笔资金部署能处理上万QPS的服务器。

1.1.3 消息分发

在大数据时代，数据对很多公司来说就像金矿，公司需要依赖对数据的分析，进行用户画像、精准推送、流程优化等各种操作，并且对处理的实时性要求越来越高。数据是不断产生的，各个分析团队、算法团队都要依赖这些数据来进行工作，这个时候有个可持久化的消息队列就非常重要。数据的产生方只需要把各自的数据写入一个消息队列即可，数据使用方根据各自需求订阅感兴趣的数据，不同数据团队所订阅的数据可以重复也可以不重复，互不干扰，也不必和数据产生方关联。

如图1-2所示，各个子系统将日志数据不停地写入消息队列，不同的数据处理系统有各自的Offset，互不影响。甚至某个团队处理完的结果数据也可以写入消息队列，作为数据的产生方，供其他团队使用，避免重复计算。在大数据时代，消息队列已经成为数据处理系统不可或缺的一部分。

除了上面列出的应用解耦、流量消峰、消息分发等功能外，消息队列还有保证最终一致性、方便动态扩容等功能。

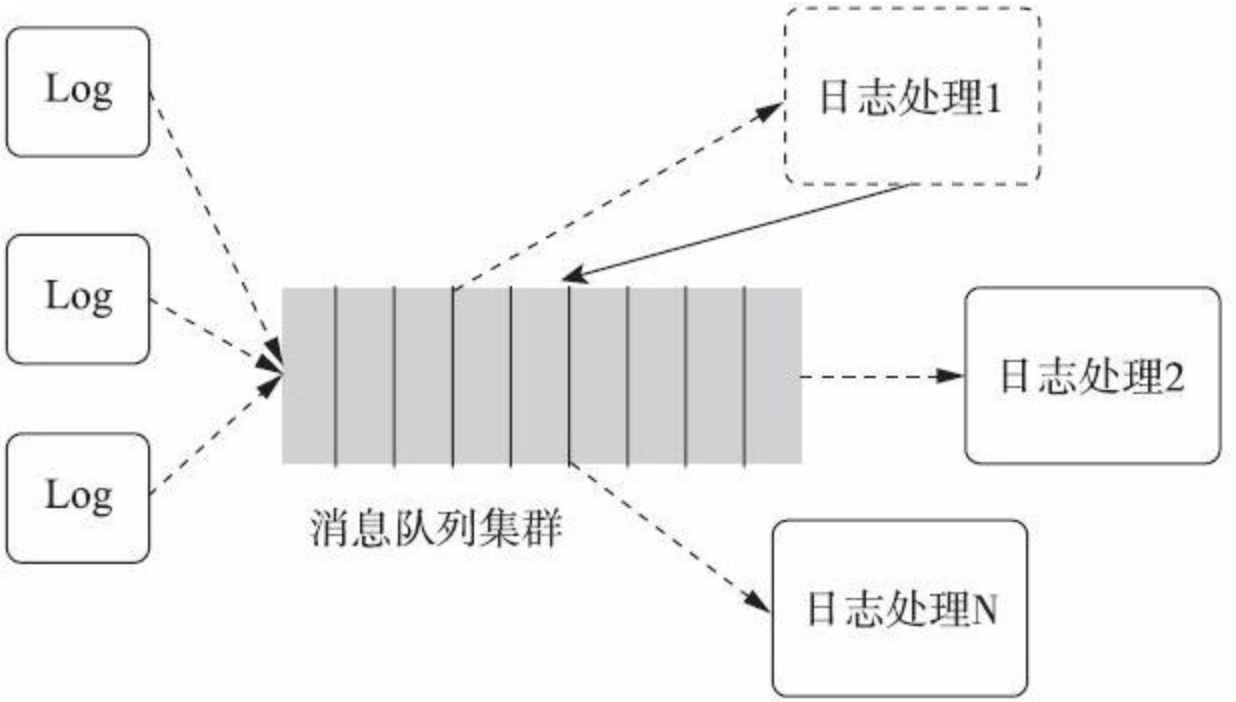


图1-2　消息队列的消息分发功能

1.2 RocketMQ简介

阿里的消息中间件有很长的历史，从2007年的Notify到2010年的Napoli，2011年升级后改为MetaQ，然后到2012年开始做RocketMQ，RocketMQ使用Java语言开发，于2016年开源。第一代的Notify主要使用了推模型，解决了事务消息；第二代的MetaQ主要使用了拉模型，解决了顺序消息和海量堆积的问题。RocketMQ基于长轮询的拉取方式，兼有两者的优点。

每一次产品迭代，都吸取了之前的经验教训，目前RocketMQ已经成为Apache顶级项目。在阿里内部，RocketMQ很好地服务了集团大大小小上千个应用，在每年的双十一当天，更有不可思议的万亿级消息通过RocketMQ流转（在2017年的双十一当天，整个阿里巴巴集团通过RocketMQ流转的线上消息达到了万亿级，峰值TPS达到5600万），在阿里大中台策略上发挥着举足轻重的作用。

此外，RocketMQ是使用Java语言开发的，比起Kafka的Scala语言和RabbitMQ的Erlang语言，更容易找到技术人员进行定制开发。

1.3 快速上手RocketMQ

本节介绍如何安装配置单机版的RocketMQ，以及简单地收发消息。读者也可以参考RocketMQ官网的说明文档。

1.3.1 RocketMQ的下载、安装和配置

RocketMQ的Binary版是一些编译好的jar和辅助的shell脚本，可以直接从官网找到下载链接（http://rocketmq.apache.org/dowloading/releases/），也可以下载源码自己编译。

系统要求：64bit的Linux、Unix或Mac。Java版本大于等于JDK1.8。如果需要从GitHub上下载源码和编译的话，需要安装Maven 3.2.x和Git。

RocketMQ当前的最新版本是4.2.0，下面以Binary版本为例说明如何快速使用：

```
> unzip rocketmq-all-4.2.0-bin-release.zip -d ./rocketmq-all-4.2.0-binls
> cd rocketmq-all-4.2.0-bin/
```

里面含有以下内容：

```
LICENSE  NOTICE  README.md  benchmark/  bin/   conf/  lib/
```

LICENSE、NOTICE和README.md包括一些版权声明和功能说明信息；benchmark里包括运行benchmark程序的shell脚本；bin文件夹里含有各种使用RocketMQ的shell脚本（Linux平台）和cmd脚本（Windows平台），比如常用的启动NameServer的脚本mqnamesrv，启动Broker的脚本mqbroker，集群管理脚本mqadmin等；conf文件夹里有一些示例配置文件，包括三种方式的broker配置文件、logback日志配置文件等，用户在写配置文件的时候，一般基于这些示例配置文件，加上自己特殊的需求即可；lib文件夹里包括RocketMQ各个模块编译成的jar包，以及RocketMQ依赖的一些jar包，比如Netty、commons-lang、FastJSON等。

1.3.2 启动消息队列服务

启动单机的消息队列服务比较简单，不需要写配置文件，只需要依次启动本机的NameServer和Broker即可。

启动NameServer：

```
> nohup sh bin/mqnamesrv &
> tail -f ~/Logs/rocketmqLogs/namesrv.Log
The Name Server boot success...
```

启动Broker：

```
> nohup sh bin/mqbroker –n localhost:9876&
> tail -f ~/Logs/rocketmqLogs/broker.Log
The broker[%s, 192.168.0.233:10911] boot success...
```

1.3.3 用命令行发送和接收消息

为了快速展示发送和接收消息，本节展示的是用命令行发送和接收消息，实际上就是运行写好的demo程序，后续我们可以参考这些demo来写自己的发送和接收程序。

运行示例程序，发送和接收消息：

```
 > export NAMESRV_ADDR=localhost:9876
 > sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer
SendResult [sendStatus=SEND_OK, msgId= ...

> sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
ConsumeMessageThread_%d Receive New Messages: [MessageExt...
```

1.3.4 关闭消息队列

消息队列被启动后，如果不主动关闭，则会一直在后台运行，占用系统资源。我们有专门用来关闭NameServer和Broker的命令。

关闭NameServer和Broker：

```
> sh bin/mqshutdown broker
The mqbroker(36695) is running...
Send shutdown request to mqbroker(36695) OK

> sh bin/mqshutdown namesrv
The mqnamesrv(36664) is running...
Send shutdown request to mqnamesrv(36664) OK
```

恭喜，现在你已经能够使用RocketMQ发送并接收消息了，使用消息队列的基本功能就是这么简单。

1.4 本章小结

本章介绍了消息队列的功能，以及RocketMQ这个消息队列从阿里诞生的历史。然后基于快速上手的目的，本章直接给出了一些命令示例，读者跟着操作即可快速启动一个RocketMQ服务，并且可以尝试发送和接收简单的消息。有了本章的初步体验后，下一章将介绍如何在生产环境使用RocketMQ。

第2章　生产环境下的配置和使用

本章的目的是带领读者快速将RocketMQ应用到生产环境中，因此不会探究原理和细节。本章会先介绍RocketMQ的各个角色，然后介绍如何搭建一个高可用的分布式消息队列集群，以及RocketMQ的Consumer和Producer的使用方法与常用命令。

2.1 RocketMQ各部分角色介绍

RocketMQ由四部分组成，先来直观地了解一下这些角色以及各自的功能。分布式消息队列是用来高效地传输消息的，它的功能和现实生活中的邮局收发信件很类似，我们类比地说一下相应的模块。现实生活中的邮政系统要正常运行，离不开下面这四个角色，一是发信者，二是收信者，三是负责暂存、传输的邮局，四是负责协调各个地方邮局的管理机构。对应到RocketMQ中，这四个角色就是Producer、Consumer、Broker和NameServer。

启动RocketMQ的顺序是先启动NameServer，再启动Broker，这时候消息队列已经可以提供服务了，想发送消息就使用Producer来发送，想接收消息就使用Consumer来接收。很多应用程序既要发送，又要接收，可以启动多个Producer和Consumer来发送多种消息，同时接收多种消息。

为了消除单点故障，增加可靠性或增大吞吐量，可以在多台机器上部署多个NameServer和Broker，为每个Broker部署一个或多个Slave。

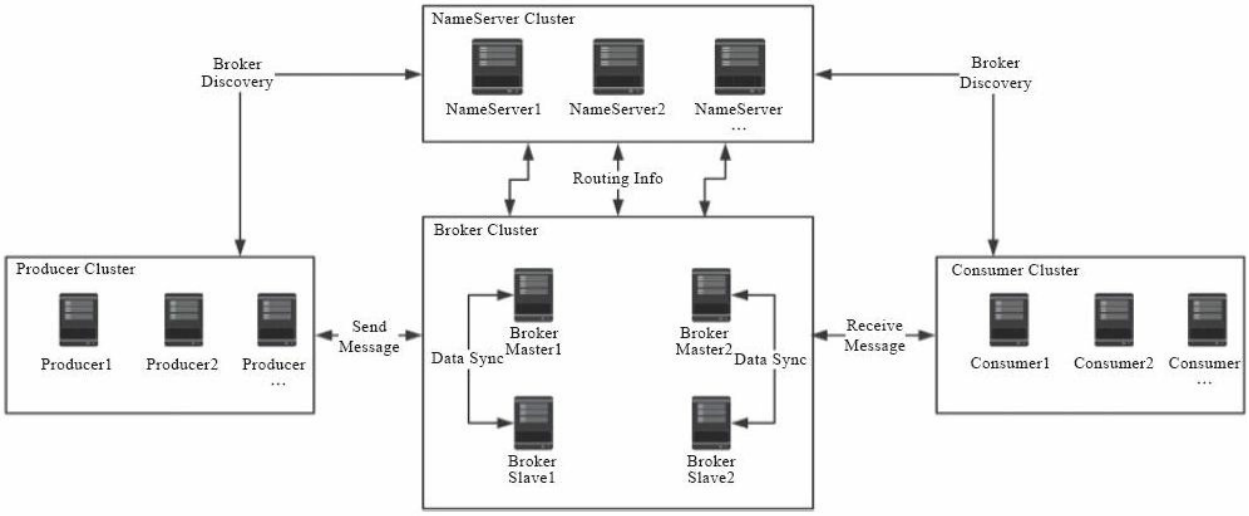


图2-1 RocketMQ各个角色间关系

了解了四种角色以后，再介绍一下Topic和Message Queue这两个名词。一个分布式消息队列中间件部署好以后，可以给很多个业务提供服务，同一个业务也有不同类型的消息要投递，这些不同类型的消息以不

同的Topic名称来区分。所以发送和接收消息前，先创建Topic，针对某个Topic发送和接收消息。有了Topic以后，还需要解决性能问题。如果一个Topic要发送和接收的数据量非常大，需要能支持增加并行处理的机器来提高处理速度，这时候一个Topic可以根据需求设置一个或多个Message Queue，Message Queue类似分区或Partition。Topic有了多个Message Queue后，消息可以并行地向各个Message Queue发送，消费者也可以并行地从多个Message Queue读取消息并消费。

2.2 多机集群配置和部署

本节将说明如何只用两台物理机，搭建出双主、双从、无单点故障的高可用RocketMQ集群。假设这两台物理机的IP分别是192.168.100.131和192.168.100.132。

2.2.1 启动多个NameServer和Broker

首先在这两台机器上分别启动NameServer（nohup sh bin/mqnamesrv &），这样我们就得到了一个无单点的NameServer服务，服务地址是“192.168.100.131：9876；192.168.100.132：9876”。

然后启动Broker，每台机器上都要分别启动一个Master角色的Broker和一个Slave角色的Broker，并互为主备。可以基于RocketMQ自带的示例配置文件写自己的配置文件（示例配置文件在conf/2m-2s-sync目录下）。

1）192.168.100.131机器上Master Broker的配置文件：

```
namesrvAddr=192.168.100.131:9876; 192.168.100.132:9876
brokerClusterName=DefaultCluster
brokerName=broker-a
brokerId=0
deleteWhen=04
fileReservedTime=48
brokerRole=SYNC_MASTER
flushDiskType=ASYNC_FLUSH
listenPort=10911
storePathRootDir=/home/rocketmq/store-a
```

2）192.168.100.132机器上Master Broker的配置文件：

```
namesrvAddr=192.168.100.131:9876; 192.168.100.132:9876
brokerClusterName=DefaultCluster
brokerName=broker-b
brokerId=0
deleteWhen=04
fileReservedTime=48
brokerRole=SYNC_MASTER
flushDiskType=ASYNC_FLUSH
listenPort=10911
storePathRootDir=/home/rocketmq/store-b
```

3）192.168.100.131机器上Slave Broker的配置文件：

```
namesrvAddr=192.168.100.131:9876; 192.168.100.132:9876
brokerClusterName=DefaultCluster
brokerName=broker-b
brokerId=1
```

```
deleteWhen=04
fileReservedTime=48
brokerRole=SLAVE
flushDiskType=ASYNC_FLUSH
listenPort=11011
storePathRootDir=/home/rocketmq/store-b
```

4）192.168.100.132机器上Slave Broker的配置文件：

```
namesrvAddr=192.168.100.131:9876; 192.168.100.132:9876
brokerClusterName=DefaultCluster
brokerName=broker-a
brokerId=1
deleteWhen=04
fileReservedTime=48
brokerRole=SLAVE
flushDiskType=ASYNC_FLUSH
listenPort=11011
storePathRootDir=/home/rocketmq/store-a
```

然后分别使用如下命令启动四个Broker：

```
nohup sh ./bin/mqbroker -c  config_file &
```

这样一个高可用的RocketMQ集群就搭建好了，还可以在一台机器上启动rocketmq-console，比如在192.168.100.131上启动RocketMQ-console，然后在浏览器中输入地址192.168.100.131：8080，这样就可以可视化地查看集群状态了。

2.2.2 配置参数介绍

本节将逐个介绍Broker配置文件中用到的参数含义：

1）namesrvAddr=192.168.100.131：9876；192.168.100.132：9876

NamerServer的地址，可以是多个。

2）brokerClusterName=DefaultCluster

Cluster的地址，如果集群机器数比较多，可以分成多个Cluster，每个Cluster供一个业务群使用。

3）brokerName=broker-a

Broker的名称，Master和Slave通过使用相同的Broker名称来表明相互关系，以说明某个Slave是哪个Master的Slave。

4）brokerId=0

一个Master Borker可以有多个Slave，0表示Master，大于0表示不同Slave的ID。

5）fileReservedTime=48

在磁盘上保存消息的时长，单位是小时，自动删除超时的消息。

6）deleteWhen=04

与fileReservedTime参数呼应，表明在几点做消息删除动作，默认值04表示凌晨4点。

7）brokerRole=SYNC_MASTER

brokerRole有3种：SYNC_MASTER、ASYNC_MASTER、SLAVE。关键词SYNC和ASYNC表示Master和Slave之间同步消息的机制，SYNC的意思是当Slave和Master消息同步完成后，再返回发送成功

的状态。

8）flushDiskType=ASYNC_FLUSH

flushDiskType表示刷盘策略，分为SYNC_FLUSH和ASYNC_FLUSH两种，分别代表同步刷盘和异步刷盘。同步刷盘情况下，消息真正写入磁盘后再返回成功状态；异步刷盘情况下，消息写入page_cache后就返回成功状态。

9）listenPort=10911

Broker监听的端口号，如果一台机器上启动了多个Broker，则要设置不同的端口号，避免冲突。

10）storePathRootDir=/home/rocketmq/store-a

存储消息以及一些配置信息的根目录。

这些配置参数，在Broker启动的时候生效，如果启动后有更改，要重启Broker。现在使用云服务或多网卡的机器比较普遍，Broker自动探测获得的ip地址可能不符合要求，通过brokerIP1=47.98.41.234这样的配置参数，可以设置Broker机器对外暴露的ip地址。

2.3 发送/接收消息示例

可以用自己熟悉的开发工具创建一个Java项目，加入RocketMQ Client包的依赖，用代码清单2-1的内容发送消息，这个示例代码是以Sync方式发送消息的。

代码清单2-1 Producer示例程序

```
public class SyncProducer {
    public static void main(String[] args) throws Exception {
        //Instantiate with a Producer group name.
        DefaultMQProducer Producer = new
            DefaultMQProducer("please_rename_unique_group_name");
        producer.setNamesrvAddr("192.168.100.131:9876");
        //Launch the instance.
        Producer.start();
        for (int i = 0; i < 100; i++) {
            //Create a Message instance, specifying Topic, tag and Message body.
            Message msg = new Message("TopicTest" /* Topic */,
                "TagA" /* Tag */,
                ("Hello RocketMQ " +
                    i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */
            );
            //Call send Message to deliver Message to one of brokers.
            SendResult sendResult = Producer.send(msg);
            System.out.printf("%s%n", sendResult);
        }
        //Shut down once the Producer instance is not longer in use.
        Producer.shutdown();
    }
}
```

主要流程是：创建一个DefaultMQProducer对象，设置好GroupName和NameServer地址后启动，然后把待发送的消息拼装成Message对象，使用Producer来发送。接下来看看如何接收消息，也就是使用DefaultMQPush-Consumer类实现的消费者程序，如代码清单2-2所示。

代码清单2-2 Consumer示例程序

```
        /*
         * Instantiate with specified Consumer group name.
         */
        DefaultMQPushConsumer Consumer = new DefaultMQPushConsumer("please rename to
        /*
         * Specify name server addresses.
        Consumer.setNamesrvAddr("192.168.249.47:9876");
        /*
```

```
         * Specify where to start in case the specified Consumer group is a brand ne
         */
Consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        //Consumer.setMessageModel(MessageModel.BROADCASTING);
        /*
         * Subscribe one more more Topics to consume.
         */
        Consumer.subscribe("TopicTest", "*");
        /*
         *  Register callback to execute on arrival of Messages fetched from brokers
         */
        Consumer.registerMessageListener(new MessageListenerConcurrently() {
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
                System.out.printf(Thread.currentThread().getName() + " Receive New M
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        /*
         *  Launch the Consumer instance.
         */
        Consumer.start();
```

Consumer或Producer都必须设置GroupName、NameServer地址以及端口号。然后指明要操作的Topic名称，最后进入发送和接收逻辑。

2.4 常用管理命令

MQAdmin是RocketMQ自带的命令行管理工具，在bin目录下，运行mqadmin即可执行。使用mqadmin命令，可以进行创建、修改Topic，更新Broker的配置信息，查询特定消息等各种操作。本节将介绍几个常用的命令。

1.创建/修改Topic

消息的发送和接收都要有对应的Topic，需要向某个Topic发送或接收消息，所以在正式使用RocketMQ进行消息发送和接收前，要先创建Topic，创建Topic的指令是updateTopic，表2-1列出了支持的参数。

表2-1 updateTopic

参数	是否必填	说明
-b	如果 -c 为空，则必填	Broker 地址，Topic 所在的 Broker(192.168.0.1:10911)

（续）

参数	是否必填	说明
-c	如果 -b 为空，则必填	Cluster 名称，表示 Topic 创建在该集群（集群可通过 clusterList 查询），如果集群中有多个 master 角色的 Broker，默认在每个 Broker 上创建 8 个读写队列
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876
-p	否	指定新 Topic 的权限限制，(2|4|6), [2:W 4:R; 6:RW]
-r	否	可读队列数（默认为 8）
-w	否	可写队列数（默认为 8）
-t	是	Topic 名称

2.删除Topic

与创建/修改Topic对应的是删除Topic，把RocketMQ系统中不用的Topic彻底清除，指令是deleteTopic，表2-2列出了支持的参数。

表2-2　deleteTopic

参数	是否必填	说明
-c	是	Cluster 名称，要删除的 Topic 所在的集群
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876; 192.168.0.2:9876
-t	是	Topic 名称

3.创建/修改订阅组

订阅组在提高系统的高可用性和吞吐量方面扮演着重要的角色，比如用Clustering模式消费一个Topic里的消息内容时，可以启动多个消费者并行消费，每个消费者只消费Topic里消息的一部分，以此提高消费速度，这个时候就是通过订阅组来指明哪些消费者是同一组，同一组的消费者共同消费同一个Topic里的内容。订阅组可以被自动创建，使用这个命令一般是用来修改订阅组，指令是updateSubGroup，表2-3列出了支持的参数。

表2-3　updateSubGroup

参数	是否必填	说明
-b	如果 -c 为空，则必填	Broker 地址，创建订阅组所在的 Broker
-c	如果 -b 为空，则必填	Cluster 名称，创建订阅组所在的 Cluster
-d	否	是否容许广播方式消费
-g	是	订阅组名
-i	否	从哪个 Broker 开始消费
-m	否	是否容许从队列的最小位置开始消费（true|false），默认会设置为 true
-q	否	消费失败的消息放到一个重试队列，每个订阅组配置的重试队列数量
-r	否	重试消费最大次数，超过则投递到死信队列
-s	否	消费功能是否开启
-w	否	发现消息堆积后，将 Consumer 的消费请求重定向到另外一台 Broker 机器
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876...

4.删除订阅组

与创建或修改订阅组相对应，这个命令删除不再使用的订阅组，指令是deleteSubGroup，表2-4列出了支持的参数。

表2-4　deleteSubGroup

参数	是否必填	说明
-b	如果 -c 为空，则必填	Broker 地址，删除订阅组所在的 Broker
-c	如果 -b 为空，则必填	Cluster 名称，删除订阅组所在的 Cluster
-g	是	订阅组名
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876...

5.更新Broker配置

Broker有很多的配置信息，在Broker启动时，可以通过配置文件来

指定配置信息。有些配置信息支持在Broker运行的时候动态更改，更改指令是updateBrokerConfig，表2-5列出了支持的参数。

表2-5　updateBrokerConfig

参数	是否必填	说明
-b	如果 -c 为空，则必填	Broker 名称
-c	如果 -b 为空，则必填	Cluster 名称，该 Broker 所在的 Cluster
-k	是	Key 值
-v	否	Value 值
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876...

6.更新Topic的读写权限

RocketMQ支持对Topic进行权限控制，主要分为只读的Topic和可读写的Topic，权限可以通过指令updateTopicPerm来动态改变，表2-6列出了支持的参数。

表2-6　updateTopicPerm

参数	是否必填	说明
-b	如果 -c 为空，则必填	Broker 地址，Topic 所在的 Broker
-c	如果 -b 为空，则必填	Cluster 名称，表示 Topic 所在的集群
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876
-p	否	指定新 Topic 的权限限制，(2|4|6), [2:W 4:R; 6:RW]
-t	是	Topic 名称

7.查询Topic的路由信息

Topic的路由信息指的是某个Topic所在的Broker相关信息，客户端可以通过NameServer来获取这些信息，本命令一般在调试的时候使用，指令是TopicRoute，表2-7列出了支持的参数。

表2-7　TopicRoute

参数	是否必填	说明
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876
-t	是	Topic 名称

8.查看Topic列表信息

上面提到的TopicRoute是列出某个Topic的相关信息，还有个指令TopicList用来列出集群中所有Topic的名称，表2-8列出了支持的参数。

表2-8　TopicList

参数	是否必填	说明
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876...

9.查看Topic统计信息

在使用RocketMQ的时候，经常需要查看某个Topic的状态，看看消息的数量，有多少未处理等，此时可以通过指令TopicStats来查询，表2-9列出了支持的参数。

表2-9　TopicStats

参数	是否必填	说明
-t	是	Topic 名称
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876...

10.根据时间查询消息

一条消息被发送到RocketMQ后，默认会带上发送的时间戳，所以我们可以根据估计的时间来查询消息，指令是printMsg，表2-10列出了支持的参数。

表2-10　printMsg

参数	是否必填	说明
-b	否	开始时间戳，格式：currentTimeMillis|yyyy-MM-dd#HH:mm:ss:SSS
-d	否	结束时间戳，格式：currentTimeMillis|yyyy-MM-dd#HH:mm:ss:SSS
-h	否	打印帮助
-t	否	Topic 名称
-s	否	Tag 名称举例：TagA || TagB
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876...

11.根据消息ID查询消息

根据消息ID可以精确定位到某条消息，但是消息ID需要通过其他方式来获取，比如可以先用时间来查询出一些消息，然后定位到要找的具体某个消息，指令是queryMsgById，表2-11列出了支持的参数。

表2-11　queryMsgById

参数	是否必填	说明
-i	是	消息 ID
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876...

12.查看集群消息

指令clusterList用来列出集群的状态，看看有哪些Broker在提供服务，表2-12列出了支持的参数。

表2-12　clusterList

参数	是否必填	说明
-m	否	是否打印更多信息

（续）

参数	是否必填	说明
-h	否	打印帮助
-n	是	NameServe 服务地址列表，举例：192.168.0.1:9876;192.168.0.2:9876...

2.5 通过图形界面管理集群

对于RocketMQ新手，可以启动运维服务，从页面上直观看到消息队列集群的状态。有一定经验以后，可以使用命令行更快捷，其功能更全面。

运维服务程序是个SpringBoot项目，需要从GitHub上的apache/rocketmq-externals里下载源码（https://github.com/apache/rocketmq-externals/tree/master/rocketmq-console ）。

进入下载源码的目录，运行如下命令即可启动：

```
mvn spring-boot:run
```

也可以编译成jar包，通过java-jar来执行。

服务启动后，在浏览器里访问server_ip_address：8080（server_ip_address是启动rocketmq-console的机器IP）地址就可看到集群的状态。

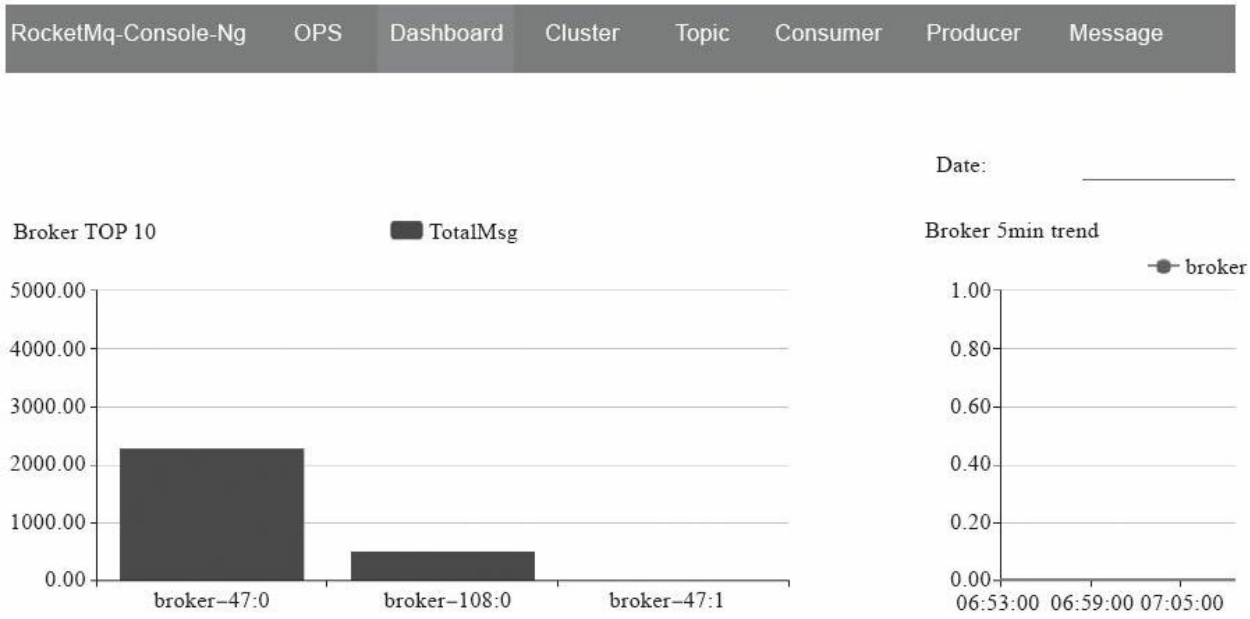


图2-2　rocketmq-console页面

2.6 本章小结

在生产环境中使用RocketMQ集群需要比QuickStart部分了解更多的内容，本章在机器角色、集群配置和部署，以及集群管理方面都做了介绍，用户可以基于这些内容搭建起一个生成环境的RocketMQ消息队列集群，在数据量不大的非关键场景，可以通过这一章快速上线。下一章重点讲如何用好RocketMQ，即根据实际场景选择合适的发送消息和接收消息的方式。

第3章　用适合的方式发送和接收消息

生产者和消费者是消息队列的两个重要角色，生产者向消息队列写入数据，消费者从消息队列里读取数据，RocketMQ的大部分用户只需要和生产者、消费者打交道。本章具体介绍不同类型生产者和消费者的特点，以及和它们相关的Offset和Log。

3.1 不同类型的消费者

根据使用者对读取操作的控制情况，消费者可分为两种类型。一个是DefaultMQPushConsumer，由系统控制读取操作，收到消息后自动调用传入的处理方法来处理；另一个是DefaultMQPullConsumer，读取操作中的大部分功能由使用者自主控制。

3.1.1 DefaultMQPushConsumer的使用

使用DefaultMQPushConsumer主要是设置好各种参数和传入处理消息的函数。系统收到消息后自动调用处理函数来处理消息，自动保存Offset，而且加入新的DefaultMQPushConsumer后会自动做负载均衡。下面结合org.apache.rocketmq.example.quickstart包中的源码来介绍，如代码清单3-1所示。

代码清单3-1　DefaultMQPushConsumer示例

```
public class QuickStart {
    public static void main(String[] args) throws InterruptedException, MQClientExce
        DefaultMQPushConsumer Consumer = new DefaultMQPushConsumer ("please_rename_u
Consumer.setNamesrvAddr("name-server1-ip:9876;name-server2-ip:9876");
Consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        Consumer.setMessageModel(MessageModel.BROADCASTING);

        Consumer.subscribe("TopicTest", "*");
        Consumer.registerMessageListener(new MessageListenerConcurrently() {
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
                System.out.printf(Thread.currentThread().getName() + " Receive New M
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        Consumer.start();
    }
}
```

DefaultMQPushConsumer需要设置三个参数：一是这个Consumer的GroupName，二是NameServer的地址和端口号，三是Topic的名称，下面将分别进行详细介绍。

1）Consumer的GroupName用于把多个Consumer组织到一起，提高并发处理能力，GroupName需要和消息模式（MessageModel）配合使用。

RocketMQ支持两种消息模式：Clustering和Broadcasting。

·在Clustering模式下，同一个ConsumerGroup（GroupName相同）里的每个Consumer只消费所订阅消息的一部分内容，同一个ConsumerGroup里所有的Consumer消费的内容合起来才是所订阅Topic内容的整体，从而达到负载均衡的目的。

·在Broadcasting模式下，同一个ConsumerGroup里的每个Consumer都能消费到所订阅Topic的全部消息，也就是一个消息会被多次分发，被多个Consumer消费。

2）NameServer的地址和端口号，可以填写多个，用分号隔开，达到消除单点故障的目的，比如“ip1：port；ip2：port；ip3：port”。

3）Topic名称用来标识消息类型，需要提前创建。如果不需要消费某个Topic下的所有消息，可以通过指定消息的Tag进行消息过滤，比如：Consumer.subscribe（"TopicTest"，"tag1||tag2||tag3"），表示这个Consumer要消费“TopicTest”下带有tag1或tag2或tag3的消息（Tag是在发送消息时设置的标签）。在填写Tag参数的位置，用null或者“*”表示要消费这个Topic的所有消息。

3.1.2 DefaultMQPushConsumer的处理流程

本节通过分析源码来说明DefaultMQPushConsumer的处理流程。

DefaultMQPushConsumer主要功能实现在DefaultMQPushConsumerImpl类中，消息的处理逻辑是在pullMessage这个函数里的PullCallBack中。在PullCallBack函数里有个switch语句，根据从Broker返回的消息类型做相应的处理，具体处理逻辑可以查看源码，如代码清单3-2所示。

代码清单3-2　DefaultMQPushConsuer的处理逻辑

```
switch (pullResult.getPullStatus()) {
    case FOUND:
        ……
        break;
    case NO_NEW_MSG:
        ……
        break;
    case OFFSET_ILLEGAL:
        ……
        break;
    default:
        break;
}
```

DefaultMQPushConsuer的源码中有很多PullRequest语句，比如Default-MQPushConsumerImpl.this.executePullRequestImmediately（pullRequest）为什么“PushConsumer”中使用“PullRequest”呢？这是通过“长轮询”方式达到Push效果的方法，长轮询方式既有Pull的优点，又兼具Push方式的实时性。

Push方式是Server端接收到消息后，主动把消息推送给Client端，实时性高。对于一个提供队列服务的Server来说，用Push方式主动推送有很多弊端：首先是加大Server端的工作量，进而影响Server的性能；其次，Client的处理能力各不相同，Client的状态不受Server控制，如果Client不能及时处理Server推送过来的消息，会造成各种潜在问题。

Pull方式是Client端循环地从Server端拉取消息，主动权在Client手

里，自己拉取到一定量消息后，处理妥当了再接着取。Pull方式的问题是循环拉取消息的间隔不好设定，间隔太短就处在一个“忙等”的状态，浪费资源；每个Pull的时间间隔太长，Server端有消息到来时，有可能没有被及时处理。

“长轮询”方式通过Client端和Server端的配合，达到既拥有Pull的优点，又能达到保证实时性的目的。我们结合源码来分析，如代码清单3-3和3-4所示。

代码清单3-3　发送Pull消息代码片段

```
PullMessageRequestHeader requestHeader = new PullMessageRequestHeader();
requestHeader.setConsumerGroup(this.ConsumerGroup);
requestHeader.setTopic(mq.getTopic());
requestHeader.setQueueId(mq.getQueueId());
requestHeader.setQueueOffset(Offset);
requestHeader.setMaxMsgNums(maxNums);
requestHeader.setSysFlag(sysFlagInner);
requestHeader.setCommitOffset(commitOffset);
requestHeader.setSuspendTimeoutMillis(brokerSuspendMaxTimeMillis);
requestHeader.setSubscription(subExpression);
requestHeader.setSubVersion(subVersion);
requestHeader.setExpressionType(expressionType);

------
PullResult pullResult = this.mQClientFactory.getMQClientAPIImpl().pullMessage(
    brokerAddr,requestHeader,timeoutMillis,communicationMode,pullCallback);
```

源码中有这一行设置语句requestHeader.setSuspendTimeoutMillis（brokerSuspendMaxTimeMillis），作用是设置Broker最长阻塞时间，默认设置是15秒，注意是Broker在没有新消息的时候才阻塞，有消息会立刻返回。

代码清单3-4　“长轮询”服务端代码片段

```
package org.apache.rocketmq.broker.longpolling
------
if (this.brokerController.getBrokerConfig().isLongPollingEnable()) {
    this.waitForRunning(5 * 1000);
} else {
    this.waitForRunning(this.brokerController.getBrokerConfig().getShortPollingTimeM
}
long beginLockTimestamp = this.systemClock.now();
this.checkHoldRequest();
long costTime = this.systemClock.now() - beginLockTimestamp;
if (costTime > 5 * 1000) {
    Log.info("[NOTIFYME] check hold request cost {} ms.", costTime);
}
```

从Broker的源码中可以看出，服务端接到新消息请求后，如果队列里没有新消息，并不急于返回，通过一个循环不断查看状态，每次waitForRunning一段时间（默认是5秒），然后后再Check。默认情况下当Broker一直没有新消息，第三次Check的时候，等待时间超过Request里面的Broker-SuspendMaxTimeMillis，就返回空结果。在等待的过程中，Broker收到了新的消息后会直接调用notifyMessageArriving函数返回请求结果。“长轮询”的核心是，Broker端HOLD住客户端过来的请求一小段时间，在这个时间内有新消息到达，就利用现有的连接立刻返回消息给Consumer。“长轮询”的主动权还是掌握在Consumer手中，Broker即使有大量消息积压，也不会主动推送给Consumer。

长轮询方式的局限性，是在HOLD住Consumer请求的时候需要占用资源，它适合用在消息队列这种客户端连接数可控的场景中。

3.1.3 DefaultMQPushConsumer的流量控制

本节分析PushConsumer的流量控制方法。PushConsumer的核心还是Pull方式，所以采用这种方式的客户端能够根据自身的处理速度调整获取消息的操作速度。因为采用多线程处理方式实现，流量控制的方面比单线程要复杂得多。

PushConsumer有个线程池，消息处理逻辑在各个线程里同时执行，这个线程池的定义如代码清单3-5所示。

代码清单3-5　DefaultMQPushConsumer的线程池定义

```
this.consumeExecutor = new ThreadPoolExecutor(
    this.defaultMQPushConsumer.getConsumeThreadMin(),
    this.defaultMQPushConsumer.getConsumeThreadMax(),
    1000 * 60,
    TimeUnit.MILLISECONDS,
    this.consumeRequestQueue,
    new ThreadFactoryImpl("ConsumeMessageThread_"));
```

Pull获得的消息，如果直接提交到线程池里执行，很难监控和控制，比如，如何得知当前消息堆积的数量？如何重复处理某些消息？如何延迟处理某些消息？RocketMQ定义了一个快照类ProcessQueue来解决这些问题，在PushConsumer运行的时候，每个Message Queue都会有个对应的ProcessQueue对象，保存了这个Message Queue消息处理状态的快照。

ProcessQueue对象里主要的内容是一个TreeMap和一个读写锁。TreeMap里以Message Queue的Offset作为Key，以消息内容的引用为Value，保存了所有从MessageQueue获取到，但是还未被处理的消息；读写锁控制着多个线程对TreeMap对象的并发访问。

有了ProcessQueue对象，流量控制就方便和灵活多了，客户端在每次Pull请求前会做下面三个判断来控制流量，如代码清单3-6所示。

代码清单3-6　PushConsumer的流量控制逻辑

```
long cachedMessageCount = processQueue.getMsgCount().get();
```

```
long cachedMessageSizeInMiB = processQueue.getMsgSize().get() / (1024 * 1024);

if (cachedMessageCount > this.defaultMQPushConsumer.getPullThresholdForQueue()) {
    this.executePullRequestLater(pullRequest, PULL_TIME_DELAY_MILLS_WHEN_FLOW_CONTRO
    if ((queueFlowControlTimes++ % 1000) == 0) {
        log.warn(
            "the cached message count exceeds the threshold {}, so do flow control,
            this.defaultMQPushConsumer.getPullThresholdForQueue(), processQueue.getM
    }
    return;
}
if (cachedMessageSizeInMiB > this.defaultMQPushConsumer.getPullThresholdSize-ForQueu
    this.executePullRequestLater(pullRequest, PULL_TIME_DELAY_MILLS_WHEN_FLOW_CONTRO
    if ((queueFlowControlTimes++ % 1000) == 0) {
        log.warn(
            "the cached message size exceeds the threshold {} MiB, so do flow contro
            this.defaultMQPushConsumer.getPullThresholdSizeForQueue(), processQueue.
    }
    return;
}
if (!this.consumeOrderly) {
    if (processQueue.getMaxSpan() > this.defaultMQPushConsumer.getConsumeConcurrentl
        this.executePullRequestLater(pullRequest, PULL_TIME_DELAY_MILLS_WHEN_FLOW_CO
        if ((queueMaxSpanFlowControlTimes++ % 1000) == 0) {
            log.warn(
                "the queue's messages, span too long, so do flow control, minOffset=
                processQueue.getMsgTreeMap().firstKey(), processQueue.getMsgTreeMap(
                pullRequest, queueMaxSpanFlowControlTimes);
        }
        return;
    }
}
```

从代码中可以看出，PushConsumer会判断获取但还未处理的消息个数、消息总大小、Offset的跨度，任何一个值超过设定的大小就隔一段时间再拉取消息，从而达到流量控制的目的。此外ProcessQueue还可以辅助实现顺序消费的逻辑。

3.1.4 DefaultMQPullConsumer

使用DefaultMQPullConsumer像使用DefaultMQPushConsumer一样需要设置各种参数，写处理消息的函数，同时还需要做额外的事情。接下来结合org.apache.rocketmq.example.simple包中的例子源码来介绍，如代码清单3-7所示。

代码清单3-7　PullConsumer示例

```
public class PullConsumer {
    private static final Map<MessageQueue, Long> OFFSE_TABLE = new HashMap<MessageQu

    public static void main(String[] args) throws MQClientException {
        DefaultMQPullConsumer Consumer = new DefaultMQPullConsumer ("please_rename_u
        Consumer.start();
        Set<MessageQueue> mqs = Consumer.fetchSubscribeMessageQueues("TopicTest1");
        for (MessageQueue mq : mqs) {
            long Offset = Consumer.fetchConsumeOffset(mq, true);
            System.out.printf("Consume from the Queue: " + mq + "%n");
            SINGLE_MQ:
            while (true) {
                try {
                    PullResult pullResult =
                        Consumer.pullBlockIfNotFound(mq, null, getMessage-QueueOffse
                    System.out.printf("%s%n", pullResult);
                    putMessageQueueOffset(mq, pullResult.getNextBegin-Offset());
                    switch (pullResult.getPullStatus()) {
                        case FOUND:
                            break;
                        case NO_MATCHED_MSG:
                            break;
                        case NO_NEW_MSG:
                            break SINGLE_MQ;
                        case OFFSET_ILLEGAL:
                            break;
                        default:
                            break;
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
        Consumer.shutdown();
    }
    private static long getMessageQueueOffset(MessageQueue mq) {
        Long Offset = OFFSE_TABLE.get(mq);
        if (Offset != null)
            return Offset;
        return 0;
    }
    private static void putMessageQueueOffset(MessageQueue mq, long Offset) {
        OFFSE_TABLE.put(mq, Offset);
    }
}
```

示例代码的处理逻辑是逐个读取某Topic下所有Message Queue的内容，读完一遍后退出，主要处理额外的三件事情：

（1）获取Message Queue并遍历

一个Topic包括多个Message Queue，如果这个Consumer需要获取Topic下所有的消息，就要遍历多有的Message Queue。如果有特殊情况，也可以选择某些特定的Message Queue来读取消息。

（2）维护Offsetstore

从一个Message Queue里拉取消息的时候，要传入Offset参数（long类型的值），随着不断读取消息，Offset会不断增长。这个时候由用户负责把Offset存储下来，根据具体情况可以存到内存里、写到磁盘或者数据库里等。

（3）根据不同的消息状态做不同的处理

拉取消息的请求发出后，会返回：FOUND、NO_MATCHED_MSG、NO_NEW_MSG、OFFSET_ILLEGAL四种状态，需要根据每个状态做不同的处理。比较重要的两个状态是FOUNT和NO_NEW_MSG，分别表示获取到消息和没有新的消息。

实际情况中可以把while（true）放到外层，达到无限循环的目的。因为PullConsumer需要用户自己处理遍历Message Queue、保存Offset，所以PullConsumer有更多的自主性和灵活性。

3.1.5 Consumer的启动、关闭流程

消息队列一般是提供一个不间断的持续性服务，Consumer在使用过程中，如何才能优雅地启动和关闭，确保不漏掉或者重复消费消息呢？

Consumer分为Push和Pull两种方式，对于PullConsumer来说，使用者主动权很高，可以根据实际需要暂停、停止、启动消费过程。需要注意的是Offset的保存，要在程序的异常处理部分增加把Offset写入磁盘方面的处理，记准了每个Message Queue的Offset，才能保证消息消费的准确性。

DefaultMQPushConsumer的退出，要调用shutdown（）函数，以便释放资源、保存Offset等。这个调用要加到Consumer所在应用的退出逻辑中。

PushConsumer在启动的时候，会做各种配置检查，然后连接NameServer获取Topic信息，启动时如果遇到异常，比如无法连接NameServer，程序仍然可以正常启动不报错（日志里有WARN信息）。在单机环境下可以测试这种情况，启动DefaultMQPushConsumer时故意把NameServer地址填错，程序仍然可以正常启动，但是不会收到消息。

为什么DefaultMQPushConsumer在无法连接NameServer时不直接报错退出呢？这和分布式系统的设计有关，RocketMQ集群可以有多个NameServer、Broker，某个机器出异常后整体服务依然可用。所以DefaultMQPushConsumer被设计成当发现某个连接异常时不立刻退出，而是不断尝试重新连接。可以进行这样一个测试，在DefaultMQPushConsumer正常运行的时候，手动kill掉Broker或NameServer，过一会儿再启动。会发现DefaultMQPushConsumer不会出错退出，在服务恢复后正常运行，在服务不可用的这段时间，仅仅会在日志里报异常信息。

如果需要在DefaultMQPushConsumer启动的时候，及时暴露配置问题，该如何操作呢？可以在Consumer.start（）语句后调用：Consumer.fetchSubscribeMe-ssageQueues（"TopicName"），这时如果配置信息写得不准确，或者当前服务不可用，这个语句会报MQClientException异常。

3.2 不同类型的生产者

生产者向消息队列里写入消息，不同的业务场景需要生产者采用不同的写入策略。比如同步发送、异步发送、延迟发送、发送事务消息等，下面具体介绍。

3.2.1 DefaultMQProducer

生产者发送消息默认使用的是DefaultMQProducer类，下面结合实际代码来详细解释，如代码清单3-8所示。

代码清单3-8 DefaultMQProduce示例

```
public class ProducerQuickStart {
    public static void main(String[] args) throws MQClientException, InterruptedExce
    DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_n
    producer.setInstanceName("instance1");
    producer.setRetryTimesWhenSendFailed(3);
producer.setNamesrvAddr("name-server1-ip:9876;name-server2-ip:9876");
        Producer.start();
        for (int i = 0; i < 1000; i++) {
            try {
                Message msg = new Message("TopicTest" /* Topic */,
                    "TagA" /* Tag */,
                    ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET)
                );
                Producer.send(msg, new SendCallback() {
                    public void onSuccess(SendResult sendResult) {
                        System.out.printf("%s%n", sendResult);
                        sendResult.getSendStatus();
                    }
                    public void onException(Throwable e) {
                        e.printStackTrace();
                    }
                });
            } catch (Exception e) {
                e.printStackTrace();
                Thread.sleep(1000);
            }
        }
        producer.shutdown();
    }
}
```

发送消息要经过五个步骤：

1）设置Producer的GroupName。

2）设置InstanceName，当一个Jvm需要启动多个Producer的时候，通过设置不同的InstanceName来区分，不设置的话系统使用默认名称“DEFAULT”。

3）设置发送失败重试次数，当网络出现异常的时候，这个次数影响消息的重复投递次数。想保证不丢消息，可以设置多重试几次。

4）设置NameServer地址。

5）组装消息并发送。

消息的发送有同步和异步两种方式，上面的代码使用的是异步方式。在第2章的例子中用的是同步方式。消息发送的返回状态有如下四种：FLUSH_DISK_TIMEOUT、FLUSH_SLAVE_TIMEOUT、SLAVE_NOT_AVAILABLE、SEND_OK，不同状态在不同的刷盘策略和同步策略的配置下含义是不同的。

·FLUSH_DISK_TIMEOUT：表示没有在规定时间内完成刷盘（需要Broker的刷盘策略被设置成SYNC_FLUSH才会报这个错误）。

·FLUSH_SLAVE_TIMEOUT：表示在主备方式下，并且Broker被设置成SYNC_MASTER方式，没有在设定时间内完成主从同步。

·SLAVE_NOT_AVAILABLE：这个状态产生的场景和FLUSH_SLAVE_TIMEOUT类似，表示在主备方式下，并且Broker被设置成SYNC_MASTER，但是没有找到被配置成Slave的Broker。

·SEND_OK：表示发送成功，发送成功的具体含义，比如消息是否已经被存储到磁盘？消息是否被同步到了Slave上？消息在Slave上是否被写入磁盘？需要结合所配置的刷盘策略、主从策略来定。这个状态还可以简单理解为，没有发生上面列出的三个问题状态就是SEND_OK。

写一个高质量的生产者程序，重点在于对发送结果的处理，要充分考虑各种异常，写清对应的处理逻辑。

3.2.2 发送延迟消息

RocketMQ支持发送延迟消息，Broker收到这类消息后，延迟一段时间再处理，使消息在规定的一段时间后生效。

延迟消息的使用方法是在创建Message对象时，调用setDelayTimeLevel（int level）方法设置延迟时间，然后再把这个消息发送出去。目前延迟的时间不支持任意设置，仅支持预设值的时间长度（1s/5s/10s/30s/1m/2m/3m/4m/5m/6m/7m/8m/9m/10m/20m/30m/1h/2h）。比如setDelayTimeLevel（3）表示延迟10s。

3.2.3 自定义消息发送规则

一个Topic会有多个Message Queue，如果使用Producer的默认配置，这个Producer会轮流向各个Message Queue发送消息。Consumer在消费消息的时候，会根据负载均衡策略，消费被分配到的Message Queue，如果不经过特定的设置，某条消息被发往哪个Message Queue，被哪个Consumer消费是未知的。

如果业务需要我们把消息发送到指定的Message Queue里，比如把同一类型的消息都发往相同的Message Queue，该怎么办呢？可以用Message-QueueSelector，如代码清单3-9所示。

代码清单3-9　MessageQueueSelector示例

```
public class OrderMessageQueueSelector implements MessageQueueSelector {
     public MessageQueue select(List<MessageQueue> mqs, Message msg,
    int id = Integer.parseInt(orderKey.toString());
    int idMainIndex  = id/100;
    int size = mqs.size();
    int index = idMainIndex%size;
    return mqs.get(index);
  }
}
```

发送消息的时候，把MessageQueueSelector的对象作为参数，使用public SendResult send（Message msg，MessageQueueSelector selector，Object arg）函数发送消息即可。在MessageQueueSelector的实现中，根据传入的Object参数，或者根据Message消息内容确定把消息发往那个Message Queue，返回被选中的Message Queue。

3.2.4 对事务的支持

RocketMQ的事务消息，是指发送消息事件和其他事件需要同时成功或同时失败。比如银行转账，A银行的某账户要转一万元到B银行的某账户。A银行发送“B银行账户增加一万元”这个消息，要和“从A银行账户扣除一万元”这个操作同时成功或者同时失败。

RocketMQ采用两阶段提交的方式实现事务消息，TransactionMQProducer处理上面情况的流程是，先发一个“准备从B银行账户增加一万元”的消息，发送成功后做从A银行账户扣除一万元的操作，根据操作结果是否成功，确定之前的“准备从B银行账户增加一万元”的消息是做commit还是rollback，具体流程如下：

1）发送方向RocketMQ发送“待确认”消息。

2）RocketMQ将收到的“待确认”消息持久化成功后，向发送方回复消息已经发送成功，此时第一阶段消息发送完成。

3）发送方开始执行本地事件逻辑。

4）发送方根据本地事件执行结果向RocketMQ发送二次确认（Commit或是Rollback）消息，RocketMQ收到Commit状态则将第一阶段消息标记为可投递，订阅方将能够收到该消息；收到Rollback状态则删除第一阶段的消息，订阅方接收不到该消息。

5）如果出现异常情况，步骤4）提交的二次确认最终未到达RocketMQ，服务器在经过固定时间段后将对“待确认”消息发起回查请求。

6）发送方收到消息回查请求后（如果发送一阶段消息的Producer不能工作，回查请求将被发送到和Producer在同一个Group里的其他Producer），通过检查对应消息的本地事件执行结果返回Commit或Roolback状态。

7）RocketMQ收到回查请求后，按照步骤4）的逻辑处理。

上面的逻辑似乎很好地实现了事务消息功能，它也是RocketMQ之前的版本实现事务消息的逻辑。但是因为RocketMQ依赖将数据顺序写到磁盘这个特征来提高性能，步骤4）却需要更改第一阶段消息的状态，这样会造成磁盘Catch的脏页过多，降低系统的性能。所以RocketMQ在4.x的版本中将这部分功能去除。系统中的一些上层Class都还在，用户可以根据实际需求实现自己的事务功能。

客户端有三个类来支持用户实现事务消息，第一个类是LocalTransaction-Executer，用来实例化步骤3）的逻辑，根据情况返回LocalTransactionState.ROLLBACK_MESSAGE或者LocalTransactionState.COMMIT_MESSAGE状态。第二个类是TransactionMQProducer，它的用法和DefaultMQProducer类似，要通过它启动一个Producer并发消息，但是比DefaultMQProducer多设置本地事务处理函数和回查状态函数。第三个类是TransactionCheckListener，实现步骤5）中MQ服务器的回查请求，返回LocalTransactionState.ROLLBACK_MESSAGE或者LocalTransactionState.COMMIT_MESSAGE。

3.3 如何存储队列位置信息

实际运行中的系统，难免会遇到重新消费某条消息、跳过一段时间内的消息等情况。这些异常情况的处理，都和Offset有关。本节主要分析Offset的存储位置，以及如何根据需要调整Offset的值。

首先来明确一下Offset的含义，RocketMQ中，一种类型的消息会放到一个Topic里，为了能够并行，一般一个Topic会有多个Message Queue（也可以设置成一个），Offset是指某个Topic下的一条消息在某个Message Queue里的位置，通过Offset的值可以定位到这条消息，或者指示Consumer从这条消息开始向后继续处理。

如图3-1所示是Offset的类结构，主要分为本地文件类型和Broker代存的类型两种。对于DefaultMQPushConsumer来说，默认是CLUSTERING模式，也就是同一个Consumer group里的多个消费者每人消费一部分，各自收到的消息内容不一样。这种情况下，由Broker端存储和控制Offset的值，使用RemoteBrokerOffsetStore结构。

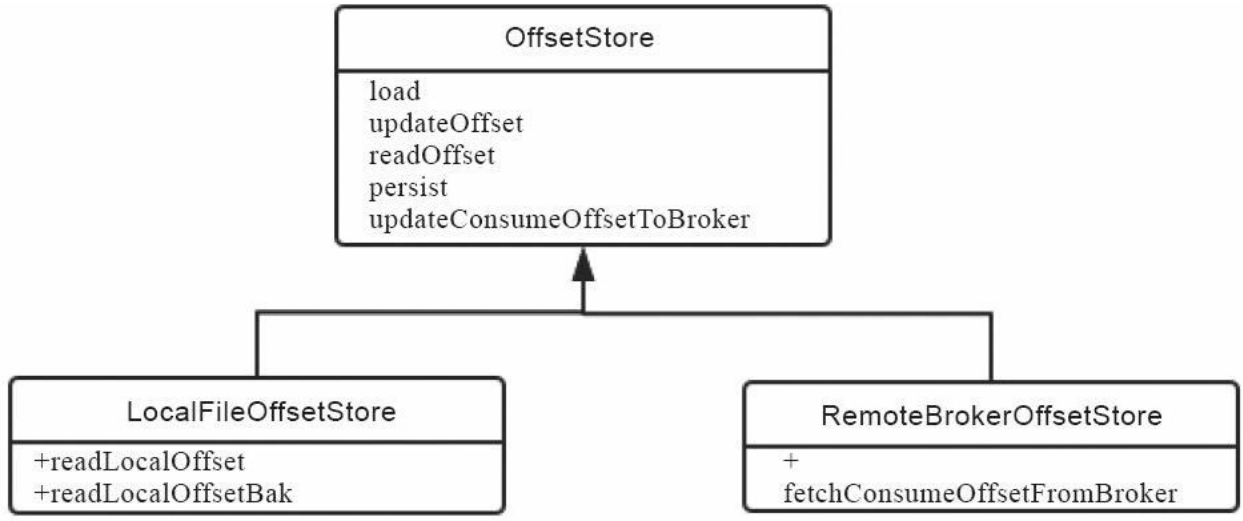


图3-1　OffsetStore的类结构

在DefaultMQPushConsumer里的BROADCASTING模式下，每个Consumer都收到这个Topic的全部消息，各个Consumer间相互没有干扰，RocketMQ使用LocalFileOffsetStore，把Offset存到本地。

OffsetStore使用Json格式存储，简洁明了，下面是个例子：

代码清单3-10　Offsetstore的内容示例

```
{"OffsetTable":{{"brokerName":"localhost", "QueueId":1,"Topic":"broker1" }: 1,{ "bro
```

在使用DefaultMQPushConsumer的时候，我们不用关心OffsetStore的事，但是如果PullConsumer，我们就要自己处理OffsetStore了。在3.1.4节的PullConsumer示例中，代码里把Offset存到了内存，没有持久化存储，这样就可能因为程序的异常或重启而丢失Offset，在实际应用中不推荐这样做。接下来给出在磁盘存储Offset的示例程序，参照LocalFileOffsetStore的源码编写，如代码清单3-11所示。

代码清单3-11　自定义持久存储OffsetStore

```
public class LocalOffsetStoreExt {
    private final String groupName;
    private final String storePath;
    private ConcurrentMap<MessageQueue, AtomicLong> OffsetTable =
            new ConcurrentHashMap<MessageQueue, AtomicLong>();
    public LocalOffsetStoreExt(String storePath, String groupName) {
        this.groupName = groupName;
        this.storePath = storePath;
    }
    public void load() {
        OffsetSerializeWrapper OffsetSerializeWrapper = this.readLocal-Offset();
        if (OffsetSerializeWrapper != null && OffsetSerializeWrapper.getOffsetTable(
            OffsetTable.putAll(OffsetSerializeWrapper.getOffsetTable());
            for (MessageQueue mq : OffsetSerializeWrapper.getOffsetTable().keySet()
                AtomicLong Offset = OffsetSerializeWrapper.getOffset-Table().get(mq
                System.out.printf("load Consumer's Offset, {} {} {} \n", this.groupN
            }
        }
    }
    public void updateOffset(MessageQueue mq, long Offset) {
        if (mq != null) {
            AtomicLong OffsetOld = this.OffsetTable.get(mq);
            if (null == OffsetOld) {
                this.OffsetTable.putIfAbsent(mq, new AtomicLong(Offset));
            } else {
                OffsetOld.set(Offset);
            }
        }
    }
    public long readOffset(final MessageQueue mq) {
        if (mq != null) {
            AtomicLong Offset = this.OffsetTable.get(mq);
            if (Offset != null) {
                return Offset.get();
            }
        }
        return 0;
    }
```

```
    public void persistAll(Set<MessageQueue> mqs) {
        if (null == mqs || mqs.isEmpty())
            return;
        OffsetSerializeWrapper OffsetSerializeWrapper = new Offset-SerializeWrapper(
        for (Map.Entry<MessageQueue, AtomicLong> entry : this.OffsetTable.
            entrySet()) {
            if (mqs.contains(entry.getKey())) {
                AtomicLong Offset = entry.getValue();
                OffsetSerializeWrapper.getOffsetTable().put(entry.getKey(), Offset);
            }
        }
        String jsonString = OffsetSerializeWrapper.toJson(true);
        if (jsonString != null) {
            try {
                MixAll.string2File(jsonString, this.storePath);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    private OffsetSerializeWrapper readLocalOffset() {
        String content = null;
        try {
            content = MixAll.file2String(this.storePath);
        } catch (IOException e) {
            e.printStackTrace();
        }
        if (null == content || content.length() == 0) {
            return null;
        } else {
            OffsetSerializeWrapper OffsetSerializeWrapper = null;
            try {
                OffsetSerializeWrapper =
                        OffsetSerializeWrapper.fromJson(content, Offset-SerializeWra
            } catch (Exception e) {
                e.printStackTrace();
            }
            return OffsetSerializeWrapper;
        }
    }
}
```

了解OffsetStore的存储机制以后，我们看看如何设置Consumer读取消息的初始位置。DefaultMQPushConsumer类里有个函数用来设置从哪儿开始消费消息：比如setConsumeFromWhere（ConsumeFromWhere.CONSUME_FROM_FIRST_这个语句设置从最小的Offset开始读取。如果从队列开始到感兴趣的消息之间有很大的范围，用CONSUME_FROM_FIRST_OFFSET参数就不合适了，可以设置从某个时间开始消费消息，比如Consumer.setConsumeF-romWhere（ConsumeFromWhere.CONSUME_FROM_TIMESTAMP），Consumer.setConsumeTimestamp（"20131223171201"），时间戳格式是精确到秒的。

注意设置读取位置不是每次都有效，它的优先级默认在Offset Store后面，比如在DefaultMQPushConsumer的BROADCASTING方式下，默认是从Broker里读取某个Topic对应ConsumerGroup的Offset，当读取不到Offset的时候，ConsumeFromWhere的设置才生效。大部分情况下这个设置在Consumer Group初次启动时有效。如果Consumer正常运行后被停止，然后再启动，会接着上次的Offset开始消费，ConsumeFromWhere的设置无效。

3.4 自定义日志输出

Log是监控系统状态，排查问题的重要手段，RocketMQ的默认Log存储位置是：${user.home}/Logs/rocketmqLogs，Log配置文件的设置可以通过JVM启动参数、环境变量、代码中的设置语句这三种方式来配置。

RocketMQ日志相关的代码在org.apache.rocketmq.Client.Log ClientLogger类中，从源码中可以看到所有的配置选项。比如想更改RocketMQ Client的Log level，可以通过-Drocketmq.Client.LogLevel来设置，或者在程序启动时使用System.setProperty（"rocketmq.Client.LogLevel"，"WARN"）来设置。

RocketMQ的Log实现是基于slf4j的，支持Logback、Log4j。RocketMQ Client里已经有Logback的相关包，可以直接使用Logback。我们可以通过Logback的配置文件对日志进行细粒度的控制。

接下来以一个maven项目为例，具体说明如何使用自定义的Log配置。

首先需要把rocketmq.Client.Log.loadconfig参数设置为false，可以在程序中使用System.setProperty（"rocketmq.Client.Log.loadconfig"，"false"）语句，或者在JVM启动时使用-D参数来设置。然后把Logback.xml放到maven项目的resources文件夹下。在Logback.xml示例配置里，在原有RocketMQ日志的基础上，增加了STDOUT输出，这样可以把RocketMQ的日志输出到应用系统console中，便于调试时发现问题，如代码清单3-12所示。

代码清单3-12　Logback.xml示例

```
<configuration>
    <appender name="RocketmqClientAppender"
              class="ch.qos.Logback.core.rolling.RollingFileAppender">
        <file>/Users/mark.yky/IdeaProjects/mqClientest/Logs/rocketmq_Client. Log</fi
        <append>true</append>
        <rollingPolicy class="ch.qos.Logback.core.rolling.FixedWindow-RollingPolicy'
            <fileNamePattern>/Users/mark.yky/IdeaProjects/mqClientest/otherdays/rock
            </fileNamePattern>
            <minIndex>1</minIndex>
```

```
            <maxIndex>20</maxIndex>
        </rollingPolicy>
        <triggeringPolicy
              class="ch.qos.Logback.core.rolling.SizeBasedTriggeringPolicy">
            <maxFileSize>100MB</maxFileSize>
        </triggeringPolicy>
        <encoder>
            <pattern>%d{yyy-MM-dd HH:mm:ss,GMT+8} %p %t - %m%n</pattern>
            <charset class="java.nio.charset.Charset">UTF-8</charset>
        </encoder>
    </appender>
    <appender name="STDOUT" class="ch.qos.Logback.core.ConsoleAppender">
        <layout class="ch.qos.Logback.classic.PatternLayout">
            <Pattern>
                %d{yyy-MM-dd HH:mm:ss,GMT+8} %p %t - %m%n
            </Pattern>
        </layout>
    </appender>
    <Logger name="RocketmqCommon" additivity="false">
        <level value="DEBUG"/>
        <appender-ref ref="RocketmqClientAppender"/>
    </Logger>
    <Logger name="RocketmqRemoting" additivity="false">
        <level value="DEBUG"/>
        <appender-ref ref="RocketmqClientAppender"/>
    </Logger>
    <Logger name="RocketmqClient" additivity="false">
        <level value="DEBUG"/>
        <appender-ref ref="RocketmqClientAppender"/>
        <appender-ref ref="STDOUT"/>
    </Logger>
</configuration>
```

有了自定义的Log配置，就可以根据实际情况，设置每个模块的输出Level，或者把日志输出到特定的位置。具体的设置方法可以参考Logback的日志配置文档：https://Logback.qos.ch/manual/configuration.html 。

3.5 本章小结

对消息队列使用者来说，Consumer和Producer是打交道最多的两个类型。本章详细介绍了两种类型的Consumer和一种类型的Producer，用户在使用的时候基于业务需求来选择合适的类型。最后重点介绍了Offset和Log，了解Offset机制是正确使用RocketMQ的基础，合理使用Log可以大幅提高开发、调试的效率。下一章将介绍RocketMQ的NameServer模块。

第4章　分布式消息队列的协调者

对于一个消息队列集群来说，系统由很多台机器组成，每个机器的角色、IP地址都不相同，而且这些信息是变动的。这种情况下，如果一个新的Producer或Consumer加入，怎么配置连接信息呢？NameServer的存在主要是为了解决这类问题，由NameServer维护这些配置信息、状态信息，其他角色都通过NameServer来协同执行。

4.1 NameServer的功能

NameServer是整个消息队列中的状态服务器，集群的各个组件通过它来了解全局的信息。同时，各个角色的机器都要定期向NameServer上报自己的状态，超时不上报的话，NameServer会认为某个机器出故障不可用了，其他的组件会把这个机器从可用列表里移除。

NamServer可以部署多个，相互之间独立，其他角色同时向多个NameServer机器上报状态信息，从而达到热备份的目的。NameServer本身是无状态的，也就是说NameServer中的Broker、Topic等状态信息不会持久存储，都是由各个角色定时上报并存储到内存中的（NameServer支持配置参数的持久化，一般用不到）。

4.1.1 集群状态的存储结构

在org.apache.rocketmq.namesrv.routeinfo的RouteInfoManager类中，有五个变量，集群的状态就保存在这五个变量中。

·private final HashMap<String/*topic*/，List<QueueData>>topicQueueTable

topicQueueTable这个结构的Key是Topic的名称，它存储了所有Topic的属性信息。Value是个QueueData队列，队里的长度等于这个Topic数据存储的Master Broker的个数，QueueData里存储着Broker的名称、读写queue的数量、同步标识等。

·private final HashMap<String/*BrokerName*/，BrokerData>Broker-AddrTable

以BrokerName为索引，相同名称的Broker可能存在多台机器，一个Master和多个Slave。这个结构存储着一个BrokerName对应的属性信息，包括所属的Cluster名称，一个Master Broker和多个Slave Broker的地址信息。

·private final HashMap<String/*ClusterName*/，Set<String/*BrokerName*/>>ClusterAddrTable

存储的是集群中Cluster的信息，结果很简单，就是一个Cluster名称对应一个由BrokerName组成的集合。

·private final HashMap<String/*BrokerAddr*/，BrokerLiveInfo>Broker-LiveTable

这个结构和BrokerAddrTable有关系，但是内容完全不同，这个结构的Key是BrokerAddr，也就是对应着一台机器，BrokerAddrTable中的Key是BrokerName，多个机器的BrokerName可以相同。BrokerLiveTable存储的内容是这台Broker机器的实时状态，包括上次更新状态的时间戳，NameServer会定期检查这个时间戳，超时没有更新就认为这个Broker无效了，将其从Broker列表里清除。

·private final HashMap<String/*BrokerAddr*/，List<String>/*Filter Server*/>filterServerTable

Filter Server是过滤服务器，是RocketMQ的一种服务端过滤方式，一个Broker可以有一个或多个Filter Server。这个结构的Key是Broker的地址，Value是和这个Broker关联的多个Filter Server的地址。

从上面这五个变量的定义，可以清楚地看出各个组件的状态是如何存储的，NameServer的主要工作就是维护这五个变量中存储的信息。

4.1.2 状态维护逻辑

本节基于源码分析NameServer如何维护各个Broker的实时状态，如何根据Broker的情况更新各种集群的属性数据。因为其他角色会主动向NameServer上报状态，所以NameServer的主要逻辑在DefaultRequest-Processor类中，根据上报消息里的请求码做相应的处理，更新存储的对应信息。此外，连接断开的事件也会触发状态更新，具体逻辑在org.apache.rocketmq.namesrv.routeinfo的BrokerHousekeepingService类中，如代码清单4-1所示。

代码清单4-1　Channel断开触发的回调函数

```
@Override
public void onChannelClose(String remoteAddr, Channel channel) {
    this.namesrvController.getRouteInfoManager().onChannelDestroy (remoteAddr, chanr
}
@Override
public void onChannelException(String remoteAddr, Channel channel) {
    this.namesrvController.getRouteInfoManager().onChannelDestroy (remoteAddr, chanr
}
@Override
public void onChannelIdle(String remoteAddr, Channel channel) {
    this.namesrvController.getRouteInfoManager().onChannelDestroy (remoteAddr, chanr
}
```

当NameServer和Broker的长连接断掉以后，onChannelDestroy函数会被调用，把这个Broker的信息清理出去。

NameServer还有定时检查时间戳的逻辑，Broker向NameServer发送的心跳会更新时间戳，当NameServer检查到时间戳长时间没有更新后，便会触发清理逻辑，如代码清单4-2所示。

代码清单4-2　定时Check Broker的状态

```
this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
    @Override
    public void run() {
        NamesrvController.this.routeInfoManager.scanNotActiveBroker();
    }
}, 5, 10, TimeUnit.SECONDS);
```

从代码可以看出是每10秒检查一次，时间戳超过2分钟则认为Broker已失效。

4.2 各个角色间的交互流程

下面从Topic的创建入手，结合源码分析一下NameServer如何和其他各个组件交互，以及NameServer存储的元数据内容的具体含义。

4.2.1 交互流程源码分析

创建Topic的代码是在org.apache.rocketmq.tools.command.topic里的UpdateTopicSubCommand类中，创建Topic的命令是updateTopic如代码清单4-3所示。

代码清单4-3 updateTopic的选项

```
Option("b", "BrokerAddr", true, "create topic to which Broker");
Option("c", "ClusterName", true, "create topic to which Cluster");
Option("t", "topic", true, "topic name");
Option("r", "readQueueNums", true, "set read queue nums");
Option("w", "writeQueueNums", true, "set write queue nums");
Option("p", "perm", true, "set topic's permission(2|4|6), intro[2:W 4:R; 6:RW]");
Option("o", "order", true, "set topic's order(true|false");
Option("u", "unit", true, "is unit topic (true|false");
Option("s", "hasUnitSub", true, "has unit sub (true|false");
```

其中b和c参数比较重要，而且他们俩只有一个会起作用（-b优先），b参数指定在哪个Broker上创建本Topic的Message Queue，c参数表示在这个Cluster下面所有的Master Broker上创建这个Topic的Message Queue，从而达到高可用性的目的。具体的创建动作是通过发送命令触发的，如代码清单4-9所示。

代码清单4-4 updateTopic的命令

```
CreateTopicRequestHeader requestHeader = new CreateTopicRequestHeader();
requestHeader.setTopic(topicConfig.getTopicName());
requestHeader.setDefaultTopic(defaultTopic);
requestHeader.setReadQueueNums(topicConfig.getReadQueueNums());
requestHeader.setWriteQueueNums(topicConfig.getWriteQueueNums());
requestHeader.setPerm(topicConfig.getPerm());
requestHeader.setTopicFilterType(topicConfig.getTopicFilterType().name());
requestHeader.setTopicSysFlag(topicConfig.getTopicSysFlag());
requestHeader.setOrder(topicConfig.isOrder());

RemotingCommand request = RemotingCommand.createRequestCommand(RequestCode. UPDATE_A
```

创建Topic的命令被发往对应的Broker，Broker接到创建Topic的请求后，执行具体的创建逻辑，如代码清单4-5所示。

代码清单4-5 Broker处理updateTopic命令

```
    private RemotingCommand updateAndCreateTopic(ChannelHandlerContext ctx, Remoting
      ...
this.BrokerController.getTopicConfigManager().updateTopicConfig(topicConfig);  //更新
        this.BrokerController.registerBrokerAll(false, true); //向NameServer发送regis
        return null;
    }
```

注意最后一步是向NameServer发送注册信息，NameServer完成创建Topic的逻辑后，其他客户端才能发现新增的Topic，相关逻辑在org.apache.rocketmq.namesrv.routeinfo的RouteInfoManager类中的registerBroker函数里，首先更新Broker信息，然后对每个Master角色的Broker，创建一个QueueData对象。如果是新建Topic，就是添加QueueData对象；如果是修改Topic，就是把旧的QueueData删除，加入新的QueueData。

4.2.2 为何不用ZooKeeper

ZooKeeper是Apache的一个开源软件，为分布式应用程序提供协调服务。那为什么RocketMQ要自己造轮子，开发集群的管理程序呢？答案是ZooKeeper的功能很强大，包括自动Master选举等，RocketMQ的架构设计决定了它不需要进行Master选举，用不到这些复杂的功能，只需要一个轻量级的元数据服务器就足够了。

中间件对稳定性要求很高，RocketMQ的NameServer只有很少的代码，容易维护，所以不需要再依赖另一个中间件，从而减少整体维护成本。

4.3 底层通信机制

分布式系统各个角色间的通信效率很关键，通信效率的高低直接影响系统性能，基于Socket实现一个高效的TCP通信协议是很有挑战的，本节介绍RocketMQ是如何解决这个问题的。

4.3.1 Remoting模块

RocketMQ的通信相关代码在Remoting模块里，先来看看主要类结构，如图4-1所示。

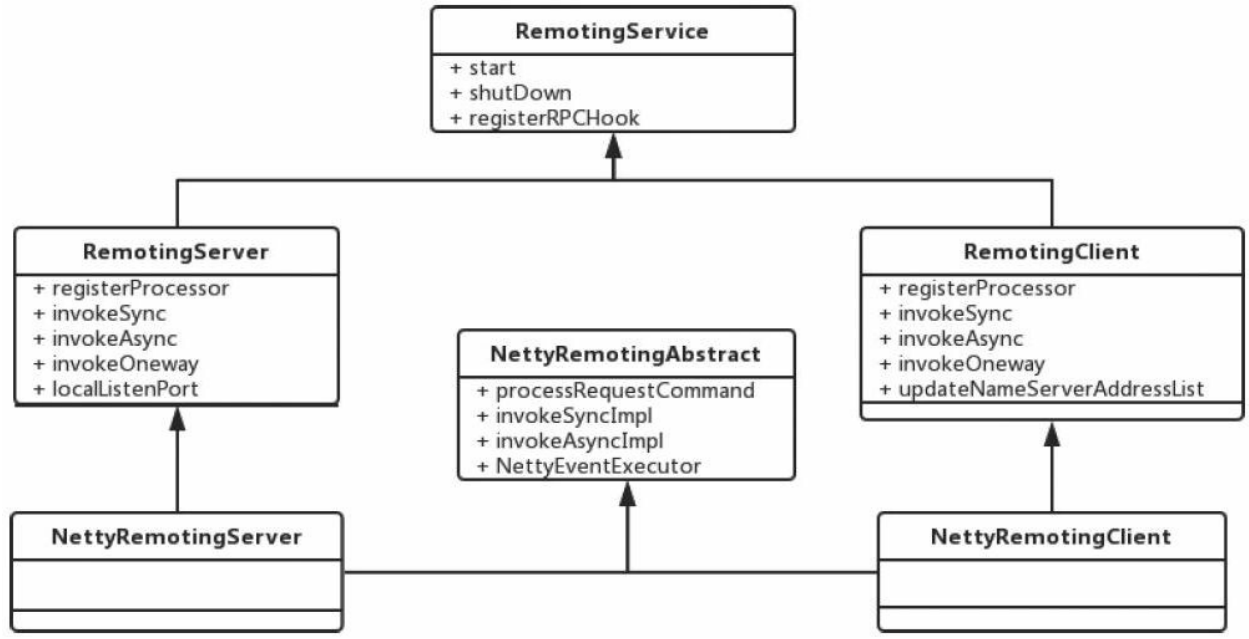


图4-1　Remoting模块的类继承关系

RemotingService为最上层接口，定义了三个方法：

·void start（）；

·void shutdown（）；

·void registerRPCHook（RPCHook rpcHook）；

RemotingClient和RemotingServer继承RemotingService接口，并增加了自己特有的方法。RemotingClient的主要函数定义如代码清单4-6所示。

代码清单4-6　RemotingClient主要函数定义

```
void registerProcessor(final int requestCode, final NettyRequestProcessor processor,
```

```
RemotingCommand invokeSync(final String addr, final RemotingCommand request, final l
void invokeAsync(final String addr, final RemotingCommand request, final long timeou
void invokeOneway(final String addr, final RemotingCommand request, final long timec
void updateNameServerAddressList(final List<String> addrs);
```

然后看看具体的实现类，NettyRemotingClient和NettyRemotingServer分别实现了RemotingClient和RemotingServer，而且都继承了NettyRemoting-Abstract类。

通过上面的封装，RocketMQ各个模块间的通信，可以通过发送统一格式的自定义消息（RemotingCommand）来完成，各个模块间的通信实现简洁明了。

比如NameServer模块中，NameServerController有一个remotingServer变量，NameServer在启动时初始化各个变量，然后启动remotingServer即可，剩下NameServer要做的是专心实现处理RemotingCommand的逻辑，如代码清单4-7所示。

代码清单4-7　NameServer处理主流程代码

```
@Override
public RemotingCommand processRequest(ChannelHandlerContext ctx, RemotingCommand rec
    if (log.isDebugEnabled()) {
        log.debug("receive request, {} {} {}",
            request.getCode(),
            RemotingHelper.parseChannelRemoteAddr(ctx.channel()),
            request);
    }
    switch (request.getCode()) {
        case RequestCode.PUT_KV_CONFIG:
            return this.putKVConfig(ctx, request);
        case RequestCode.GET_KV_CONFIG:
            return this.getKVConfig(ctx, request);
        case RequestCode.DELETE_KV_CONFIG:
            return this.deleteKVConfig(ctx, request);
        case RequestCode.REGISTER_BROKER:
            Version brokerVersion = MQVersion.value2Version(request.getVersion());
            if (brokerVersion.ordinal() >= MQVersion.Version.V3_0_11.ordinal()) {
                return this.registerBrokerWithFilterServer(ctx, request);
            } else {
                return this.registerBroker(ctx, request);
            }
        case RequestCode.UNREGISTER_BROKER:
            return this.unregisterBroker(ctx, request);
        case RequestCode.GET_ROUTEINTO_BY_TOPIC:
            return this.getRouteInfoByTopic(ctx, request);
        case RequestCode.GET_BROKER_CLUSTER_INFO:
            return this.getBrokerClusterInfo(ctx, request);
        case RequestCode.WIPE_WRITE_PERM_OF_BROKER:
            return this.wipeWritePermOfBroker(ctx, request);
        case RequestCode.GET_ALL_TOPIC_LIST_FROM_NAMESERVER:
            return getAllTopicListFromNameserver(ctx, request);
```

```
            case RequestCode.DELETE_TOPIC_IN_NAMESRV:
                return deleteTopicInNamesrv(ctx, request);
            case RequestCode.GET_KVLIST_BY_NAMESPACE:
                return this.getKVListByNamespace(ctx, request);
            case RequestCode.GET_TOPICS_BY_CLUSTER:
                return this.getTopicsByCluster(ctx, request);
            case RequestCode.GET_SYSTEM_TOPIC_LIST_FROM_NS:
                return this.getSystemTopicListFromNs(ctx, request);
            case RequestCode.GET_UNIT_TOPIC_LIST:
                return this.getUnitTopicList(ctx, request);
            case RequestCode.GET_HAS_UNIT_SUB_TOPIC_LIST:
                return this.getHasUnitSubTopicList(ctx, request);
            case RequestCode.GET_HAS_UNIT_SUB_UNUNIT_TOPIC_LIST:
                return this.getHasUnitSubUnUnitTopicList(ctx, request);
            case RequestCode.UPDATE_NAMESRV_CONFIG:
                return this.updateConfig(ctx, request);
            case RequestCode.GET_NAMESRV_CONFIG:
                return this.getConfig(ctx, request);
            default:
                break;
        }
        return null;
    }
```

在Consumer的源码中，获取消息的底层通信部分同样发送一个RemotingCommand请求，返回的response也是个RemotingCommand类型，如代码清单4-8所示。

代码清单4-8　Consumer请求消息底层实现代码

```
    private PullResult pullMessageSync(//
        final String addr, // 1
        final RemotingCommand request, // 2
        final long timeoutMillis// 3
    ) throws RemotingException, InterruptedException, MQBrokerException {
        RemotingCommand response = this.remotingClient.invokeSync(addr, request, timeout
        assert response != null;
        return this.processPullResponse(response);
    }
```

从源码中可以看出，RocketMQ中复杂的通信过程，被RemotingCommand统一起来，大部分的逻辑都是通过发送、接受并处理Command来完成的。

4.3.2 协议设计和编解码

RocketMQ自己定义了一个通信协议，使得模块间传输的二进制消息和有意义的内容之间互相转换。协议格式如图4-2所示。

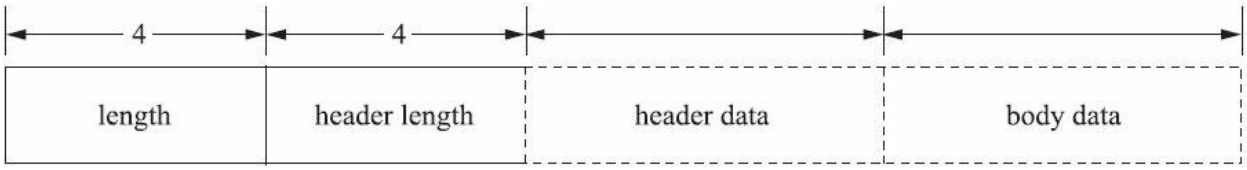


图4-2 RocketMQ的通信协议

1）第一部分是大端4个字节整数，值等于第二、三、四部分长度的总和；

2）第二部分是大端4个字节整数，值等于第三部分的长度；

3）第三部分是通过Json序列化的数据；

4）第四部分是通过应用自定义二进制序列化的数据。

消息的解码过程在RemotingCommand的decode函数里，如代码清单4-9所示。

代码清单4-9 消息解码函数

```
public static RemotingCommand decode(final ByteBuffer byteBuffer) {
    int length = byteBuffer.limit();
    int oriHeaderLen = byteBuffer.getInt();
    int headerLength = getHeaderLength(oriHeaderLen);
    byte[] headerData = new byte[headerLength];
    byteBuffer.get(headerData);
    RemotingCommand cmd = headerDecode(headerData, getProtocolType (oriHeaderLen));
    int bodyLength = length - 4 - headerLength;
    byte[] bodyData = null;
    if (bodyLength > 0) {
        bodyData = new byte[bodyLength];
        byteBuffer.get(bodyData);
    }
    cmd.body = bodyData;
    return cmd;
}
```

对应的消息编码过程在RemotingCommand的encode函数中，如代码清单4-10所示。

代码清单4-10　消息编码函数

```
public ByteBuffer encode() {
    // 1> header length size
    int length = 4;
    // 2> header data length
    byte[] headerData = this.headerEncode();
    length += headerData.length;
    // 3> body data length
    if (this.body != null) {
        length += body.length;
    }
    ByteBuffer result = ByteBuffer.allocate(4 + length);
    // length
    result.putInt(length);
    // header length
    result.put(markProtocolType(headerData.length, serializeTypeCurrentRPC));
    // header data
    result.put(headerData);
    // body data;
    if (this.body != null) {
        result.put(this.body);
    }
    result.flip();
    return result;
}
```

4.3.3 Netty库

RocketMQ是基于Netty库来完成RemotingServer和RemotingClient具体的通信实现的，Netty是个事件驱动的网络编程框架，它屏蔽了Java Socket、NIO等复杂细节，用户只需用好Netty，就可以实现一个“网络编程专家+并发编程专家”水平的Server、Client网络程序。应用Netty有一定的门槛，需要了解它的EventLoopGroup、Channel、Handler模型以及各种具体的配置。RocketMQ利用Netty实现的通信类是NettyRemotingServer和NettyRemotingClient，用户也可以参考这两个类的实现来学习使用Netty。

4.4 本章小结

本章介绍了NameServer的功能，NameServer在RocketMQ集群中扮演调度中心的角色。各个Producer、Consumer上报自己的状态上去，同时从NameServer获取其他角色的状态信息。NameServer的功能虽然非常重要，但是被设计得很轻量级，代码量少并且几乎无磁盘存储，所有的功能都通过内存高效完成。本章还介绍了底层的通信机制，RocketMQ基于Netty对底层通信做了很好的抽象，使得通信功能逻辑清晰，代码简单。Netty的介绍和具体的通信实现可以查看第13章。

第5章　消息队列的核心机制

Broker是RocketMQ的核心，大部分“重量级”工作都是由Broker完成的，包括接收Producer发过来的消息、处理Consumer的消费消息请求、消息的持久化存储、消息的HA机制以及服务端过滤功能等。

5.1 消息存储和发送

分布式队列因为有高可靠性的要求，所以数据要通过磁盘进行持久化存储。用磁盘存储消息，速度会不会很慢呢？能满足实时性和高吞吐量的要求吗？

实际上，磁盘有时候会比你想象的快很多，有时候也会比你想象的慢很多，关键在如何使用，使用得当，磁盘的速度完全可以匹配上网络的数据传输速度。目前的高性能磁盘，顺序写速度可以达到600MB/s，超过了一般网卡的传输速度，这是磁盘比想象的快的地方。但是磁盘随机写的速度只有大概100KB/s，和顺序写的性能相差6000倍！因为有如此巨大的速度差别，好的消息队列系统会比普通的消息队列系统速度快多个数量级。

举个例子，Linux操作系统分为“用户态”和“内核态”，文件操作、网络操作需要涉及这两种形态的切换，免不了进行数据复制，一台服务器把本机磁盘文件的内容发送到客户端，一般分为两个步骤：

1）read（file，tmp_buf，len）；，读取本地文件内容；

2）write（socket，tmp_buf，len）；，将读取的内容通过网络发送出去。

tmp_buf是预先申请的内存，这两个看似简单的操作，实际进行了4次数据复制，分别是：从磁盘复制数据到内核态内存，从内核态内存复制到用户态内存（完成了read（file，tmp_buf，len））；然后从用户态内存复制到网络驱动的内核态内存，最后是从网络驱动的内核态内存复制到网卡中进行传输（完成write（socket，tmp_buf，len））。

通过使用mmap的方式，可以省去向用户态的内存复制，提高速度。这种机制在Java中是通过MappedByteBuffer实现的，具体可以参考Java 7的文档：https://docs.oracle.com/javase/7/docs/api/java/nio/MappedByteBuffer.ht。RocketMQ充分利用了上述特性，也就是所谓的“零拷贝”技术，提高消息存盘和网络发送的速度。

5.2 消息存储结构

RocketMQ的具体消息存储结构是怎样的呢？如何尽量保证顺序写的呢？先来看看整体的架构图，如图5-1所示。

RocketMQ消息的存储是由ConsumeQueue和CommitLog配合完成的，消息真正的物理存储文件是CommitLog，ConsumeQueue是消息的逻辑队列，类似数据库的索引文件，存储的是指向物理存储的地址。每个Topic下的每个Message Queue都有一个对应的ConsumeQueue文件。文件地址在${$storeRoot}\consumequeue\${topicName}\${queueId}\${fileName}。

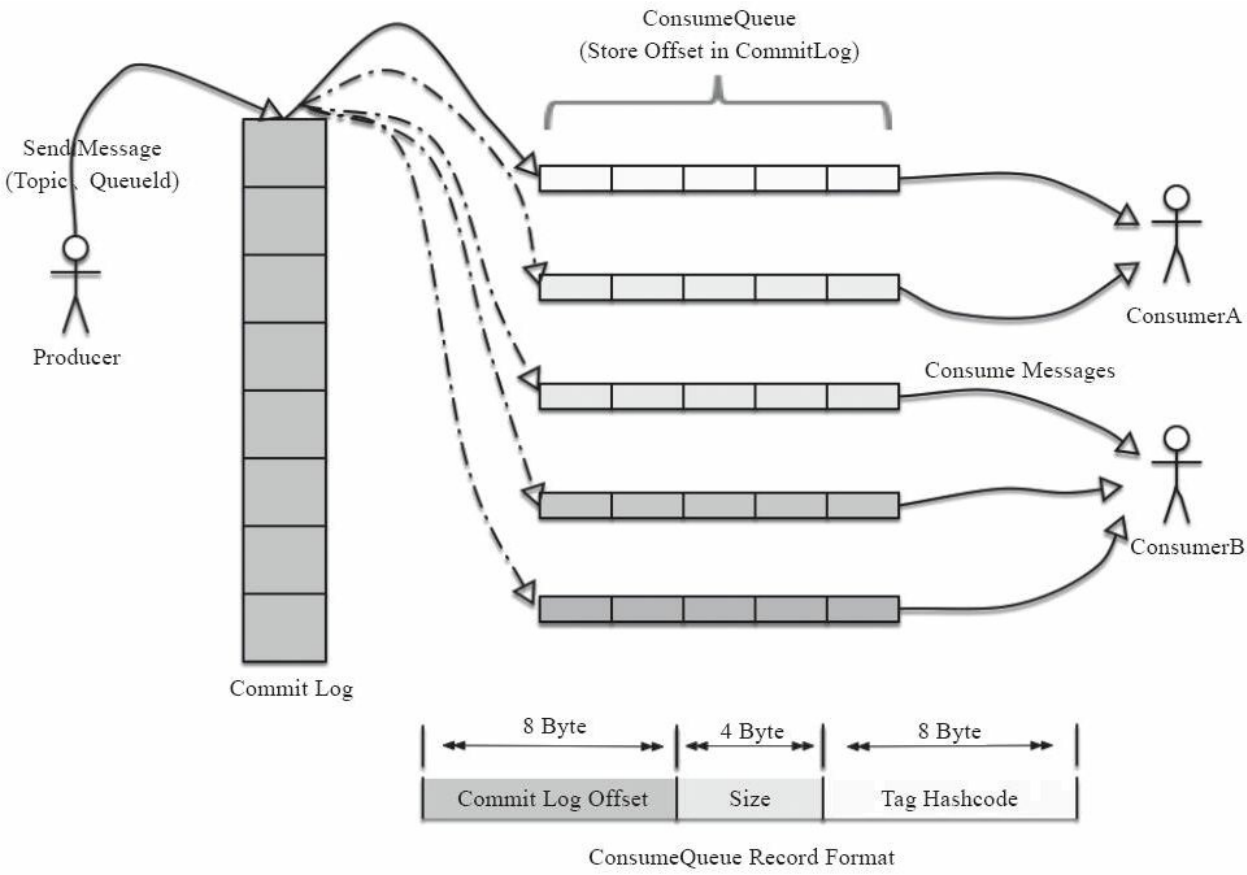


图5-1　RocketMQ的存储结构图

CommitLog以物理文件的方式存放，每台Broker上的CommitLog被本机器所有ConsumeQueue共享，文件地址：

${user.home}\store\${commitlog}\${fileName}。在CommitLog中，一个消息的存储长度是不固定的，RocketMQ采取一些机制，尽量向CommitLog中顺序写，但是随机读。ConsumeQueue的内容也会被写到磁盘里作持久存储。

存储机制这样设计有以下几个好处：

1）CommitLog顺序写，可以大大提高写入效率。

2）虽然是随机读，但是利用操作系统的pagecache机制，可以批量地从磁盘读取，作为cache存到内存中，加速后续的读取速度。

3）为了保证完全的顺序写，需要ConsumeQueue这个中间结构，因为ConsumeQueue里只存偏移量信息，所以尺寸是有限的，在实际情况中，大部分的ConsumeQueue能够被全部读入内存，所以这个中间结构的操作速度很快，可以认为是内存读取的速度。此外为了保证CommitLog和ConsumeQueue的一致性，CommitLog里存储了Consume Queues、Message Key、Tag等所有信息，即使ConsumeQueue丢失，也可以通过commitLog完全恢复出来。

如图5-2所示是一个Broker在文件系统中存储的各个文件。我们可以看到commitlog文件夹、consumequeue文件夹，还有在config文件夹中Topic、Consumer的相关信息。最下面那个文件夹index存的是索引文件，这个文件用来加快消息查询的速度。

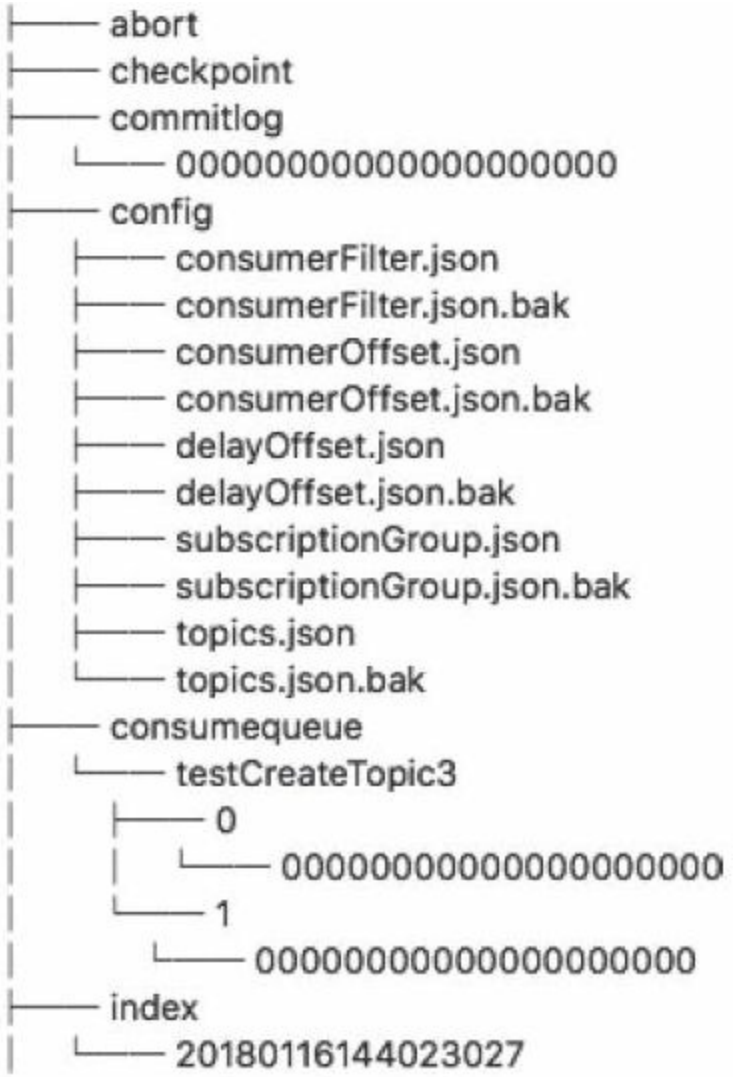


图5-2　RocketMQ的Broker机器磁盘上的文件存储结构

5.3 高可用性机制

RocketMQ分布式集群是通过Master和Slave的配合达到高可用性的，首先说一下Master和Slave的区别：在Broker的配置文件中，参数brokerId的值为0表明这个Broker是Master，大于0表明这个Broker是Slave，同时brokerRole参数也会说明这个Broker是Master还是Slave。Master角色的Broker支持读和写，Slave角色的Broker仅支持读，也就是Producer只能和Master角色的Broker连接写入消息；Consumer可以连接Master角色的Broker，也可以连接Slave角色的Broker来读取消息。

在Consumer的配置文件中，并不需要设置是从Master读还是从Slave读，当Master不可用或者繁忙的时候，Consumer会被自动切换到从Slave读。有了自动切换Consumer这种机制，当一个Master角色的机器出现故障后，Consumer仍然可以从Slave读取消息，不影响Consumer程序。这就达到了消费端的高可用性。

如何达到发送端的高可用性呢？在创建Topic的时候，把Topic的多个Message Queue创建在多个Broker组上（相同Broker名称，不同brokerId的机器组成一个Broker组），这样当一个Broker组的Master不可用后，其他组的Master仍然可用，Producer仍然可以发送消息。RocketMQ目前还不支持把Slave自动转成Master，如果机器资源不足，需要把Slave转成Master，则要手动停止Slave角色的Broker，更改配置文件，用新的配置文件启动Broker。

5.4 同步刷盘和异步刷盘

RocketMQ的消息是存储到磁盘上的，这样既能保证断电后恢复，又可以让存储的消息量超出内存的限制。RocketMQ为了提高性能，会尽可能地保证磁盘的顺序写。消息在通过Producer写入RocketMQ的时候，有两种写磁盘方式，下面逐一介绍。

·异步刷盘方式：在返回写成功状态时，消息可能只是被写入了内存的PAGECACHE，写操作的返回快，吞吐量大；当内存里的消息量积累到一定程度时，统一触发写磁盘动作，快速写入。

·同步刷盘方式：在返回写成功状态时，消息已经被写入磁盘。具体流程是，消息写入内存的PAGECACHE后，立刻通知刷盘线程刷盘，然后等待刷盘完成，刷盘线程执行完成后唤醒等待的线程，返回消息写成功的状态。

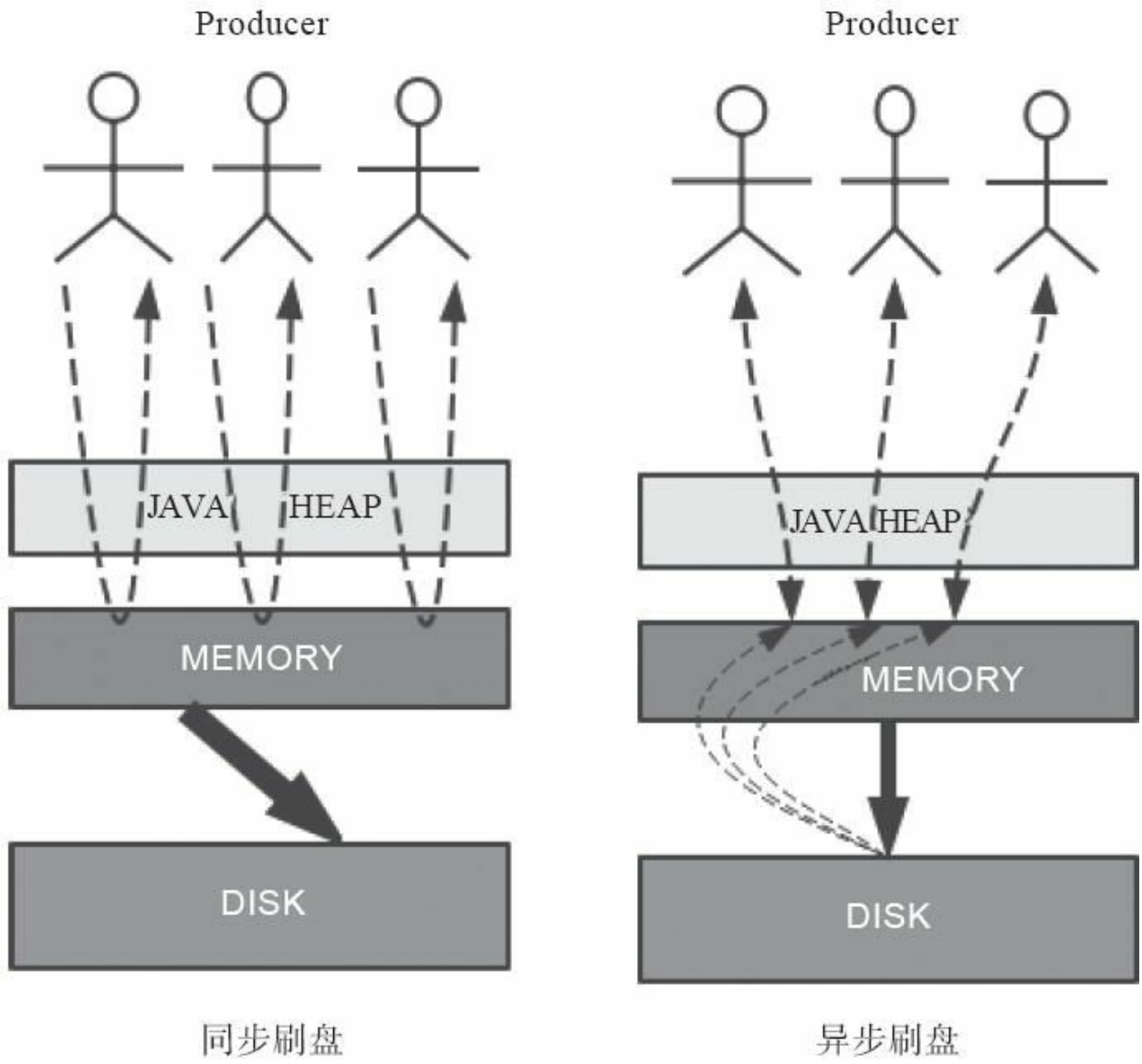


图5-3　同步刷盘和异步刷盘

同步刷盘还是异步刷盘，是通过Broker配置文件里的flushDiskType参数设置的，这个参数被配置成SYNC_FLUSH、ASYNC_FLUSH中的一个。

5.5 同步复制和异步复制

如果一个Broker组有Master和Slave，消息需要从Master复制到Slave上，有同步和异步两种复制方式。同步复制方式是等Master和Slave均写成功后才反馈给客户端写成功状态；异步复制方式是只要Master写成功即可反馈给客户端写成功状态。

这两种复制方式各有优劣，在异步复制方式下，系统拥有较低的延迟和较高的吞吐量，但是如果Master出了故障，有些数据因为没有被写入Slave，有可能会丢失；在同步复制方式下，如果Master出故障，Slave上有全部的备份数据，容易恢复，但是同步复制会增大数据写入延迟，降低系统吞吐量。

同步复制和异步复制是通过Broker配置文件里的brokerRole参数进行设置的，这个参数可以被设置成ASYNC_MASTER、SYNC_MASTER、SLAVE三个值中的一个。

实际应用中要结合业务场景，合理设置刷盘方式和主从复制方式，尤其是SYNC_FLUSH方式，由于频繁地触发磁盘写动作，会明显降低性能。通常情况下，应该把Master和Save配置成ASYNC_FLUSH的刷盘方式，主从之间配置成SYNC_MASTER的复制方式，这样即使有一台机器出故障，仍然能保证数据不丢，是个不错的选择。

5.6 本章小结

本章介绍了RocketMQ消息队列实现的难点及核心，即“队列”本身的实现，基于磁盘做一个读写效率高的队列并非易事，实现不好就会使磁盘操作成为整个系统的瓶颈，无法提升系统的吞吐量。RocketMQ基于“顺序写”“随机读”的原则来设计，利用“零拷贝”技术，克服了磁盘操作的瓶颈。

另一个难点是为了高可用性而设计的主从机制，数据被及时复制到多个机器，这样当一台机器出故障后，整体系统依然可用。这样可靠性和性能能直接有个权衡，RocketMQ把选择权留给用户，用户根据具体的业务场景来选择要更高的可靠性，还是要更高的效率。

第6章　可靠性优先的使用场景

本章的重点是可靠性，解决如何让消息队列满足业务逻辑需求，同时稳定、可靠地长期运行。

6.1 顺序消息

顺序消息是指消息的消费顺序和产生顺序相同，在有些业务逻辑下，必须保证顺序。比如订单的生成、付款、发货，这3个消息必须按顺序处理才行。顺序消息分为全局顺序消息和部分顺序消息，全局顺序消息指某个Topic下的所有消息都要保证顺序；部分顺序消息只要保证每一组消息被顺序消费即可，比如上面订单消息的例子，只要保证同一个订单ID的三个消息能按顺序消费即可。

6.1.1 全局顺序消息

RocketMQ在默认情况下不保证顺序，比如创建一个Topic，默认八个写队列，八个读队列。这时候一条消息可能被写入任意一个队列里；在数据的读取过程中，可能有多个Consumer，每个Consumer也可能启动多个线程并行处理，所以消息被哪个Consumer消费，被消费的顺序和写入的顺序是否一致是不确定的。

要保证全局顺序消息，需要先把Topic的读写队列数设置为一，然后Producer和Consumer的并发设置也要是一。简单来说，为了保证整个Topic的全局消息有序，只能消除所有的并发处理，各部分都设置成单线程处理。这时高并发、高吞吐量的功能完全用不上了。

在实际应用中，更多的是像订单类消息那样，只需要部分有序即可。在这种情况下，我们经过合适的配置，依然可以利用RocketMQ高并发、高吞吐量的能力。

6.1.2 部分顺序消息

要保证部分消息有序，需要发送端和消费端配合处理。在发送端，要做到把同一业务ID的消息发送到同一个Message Queue；在消费过程中，要做到从同一个Message Queue读取的消息不被并发处理，这样才能达到部分有序。

发送端使用MessageQueueSelector类来控制把消息发往哪个Message Queue，如代码清单6-1所示。

代码清单6-1 MessageQueueSelector示例

```
for (int i = 0; i < 100; i++) {
    int orderId = i;
    //Create a message instance, specifying topic, tag and message body.
    Message msg = new Message("OrderTopic8", tags, "KEY" + i,
        ("Hello RocketMQ " +orderId+"  "+ i).getBytes(RemotingHelper.DEFAULT_CHARSET
    SendResult sendResult = Producer.send(msg, new MessageQueueSelector() {
        @Override
        public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg)
            System.out.println("queue selector mq nums:"+mqs.size());
            System.out.println("msg info:"+msg.toString());
            for(MessageQueue mq: mqs){
                System.out.println(mq.toString());
            }
            Integer id = (Integer) arg;
            int index = id % mqs.size();
            return mqs.get(index);
        }
    }, orderId);
    System.out.println(sendResult);
}
```

消费端通过使用MessageListenerOrderly类来解决单Message Queue的消息被并发处理的问题，如代码清单6-2所示。

代码清单6-2 MessageListenerOrderly示例

```
consumer.registerMessageListener(new MessageListenerOrderly() {
    AtomicLong consumeTimes = new AtomicLong(0);
    @Override
    public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs,
                                ConsumeOrderlyContext context) {
        System.out.printf(" Received New Messages: " + new String(msgs.get(0).getBod
            return ConsumeOrderlyStatus.SUCCESS;
    }
```

```
});
```

Consumer使用MessageListenerOrderly的时候，下面四个Consumer的设置依旧可以使用：setConsumeThreadMin、setConsumeThreadMax、setPull-BatchSize、setConsumeMessageBatchMaxSize。前两个参数设置Consumer的线程数，PullBatchSize指的是一次从Broker的一个Message Queue获取消息的最大数量，默认值是32，ConsumeMessageBatchMaxSize指的是这个Consumer的Executor（也就是调用MessageListener处理的地方）一次传入的消息数（List<MessageExt>msgs这个链表的最大长度），默认值是1。

上述四个参数可以使用，说明MessageListenerOrderly并不是简单地禁止并发处理。在MessageListenerOrderly的实现中，为每个Consumer Queue加个锁，消费每个消息前，需要先获得这个消息对应的Consumer Queue所对应的锁，这样保证了同一时间，同一个Consumer Queue的消息不被并发消费，但不同Consumer Queue的消息可以并发处理。

6.2 消息重复问题

对分布式消息队列来说，同时做到确保一定投递和不重复投递是很难的，也就是所谓的“有且仅有一次”。在鱼和熊掌不可兼得的情况下，RocketMQ选择了确保一定投递，保证消息不丢失，但有可能造成消息重复。

消息重复一般情况下不会发生，但是如果消息量大，网络有波动，消息重复就是个大概率事件。比如Producer有个函数setRetryTimesWhenSendFailed，设置在同步方式下自动重试的次数，默认值是2，这样当第一次发送消息时，Broker端接收到了消息但是没有正确返回发送成功的状态，就造成了消息重复。

解决消息重复有两种方法：第一种方法是保证消费逻辑的幂等性（多次调用和一次调用效果相同）；另一种方法是维护一个已消费消息的记录，消费前查询这个消息是否被消费过。这两种方法都需要使用者自己实现。

6.3 动态增减机器

一个消息队列集群由多台机器组成，持续稳定地提供服务，因为业务需求或硬件故障，经常需要增加或减少各个角色的机器，本节介绍如何在不影响服务稳定性的情况下动态地增减机器。

6.3.1 动态增减NameServer

NameServer是RocketMQ集群的协调者，集群的各个组件是通过NameServer获取各种属性和地址信息的。主要功能包括两部分：一个各个Broker定期上报自己的状态信息到NameServer；另一个是各个客户端，包括Producer、Consumer，以及命令行工具，通过NameServer获取最新的状态信息。所以，在启动Broker、生产者和消费者之前，必须告诉它们NameServer的地址，为了提高可靠性，建议启动多个NameServer。NameServer占用资源不多，可以和Broker部署在同一台机器。有多个NameServer后，减少某个NameServer不会对其他组件产生影响。

有四种种方式可设置NameServer的地址，下面按优先级由高到低依次介绍：

1）通过代码设置，比如在Producer中，通过Producer.setNamesrvAddr（"name-server1-ip：port；name-server2-ip：port"）来设置。在mqadmin命令行工具中，是通过-n name-server-ip1：port；name-server-ip2：port参数来设置的，如果自定义了命令行工具，也可以通过defaultMQAdminExt.setNamesrvAddr（"name-server1-ip：port；name-server2-ip：port"）来设置。

2）使用Java启动参数设置，对应的option是rocketmq.namesrv.addr。

3）通过Linux环境变量设置，在启动前设置变量：NAMESRV_ADDR。

4）通过HTTP服务来设置，当上述方法都没有使用，程序会向一个HTTP地址发送请求来获取NameServer地址，默认的URL是http://jmenv.tbsite.net:8080/rocketmq/nsaddr （淘宝的测试地址），通过rocketmq.namesrv.domain参数来覆盖jmenv.tbsite.net；通过rocketmq.namesrv.domain.subgroup参数来覆盖nsaddr。

第4种方式看似繁琐，但它是唯一支持动态增加NameServer，无须重启其他组件的方式。使用这种方式后其他组件会每隔2分钟请求一次

该URL，获取最新的NameServer地址。

6.3.2 动态增减Broker

由于业务增长，需要对集群进行扩容的时候，可以动态增加Broker角色的机器。只增加Broker不会对原有的Topic产生影响，原来创建好的Topic中数据的读写依然在原来的那些Broker上进行。

集群扩容后，一是可以把新建的Topic指定到新的Broker机器上，均衡利用资源；另一种方式是通过updateTopic命令更改现有的Topic配置，在新加的Broker上创建新的队列。比如TestTopic是现有的一个Topic，因为数据量增大需要扩容，新增的一个Broker机器地址是192.168.0.1：10911，这个时候执行下面的命令：sh./bin/mqadmin updateTopic-b 192.168.0.1：10911-t TestTopic-n 192.168.0.100：9876，结果是在新增的Broker机器上，为TestTopic新创建了8个读写队列。

如果因为业务变动或者置换机器需要减少Broker，此时该如何操作呢？减少Broker要看是否有持续运行的Producer，当一个Topic只有一个Master Broker，停掉这个Broker后，消息的发送肯定会受到影响，需要在停止这个Broker前，停止发送消息。

当某个Topic有多个Master Broker，停了其中一个，这时候是否会丢失消息呢？答案和Producer使用的发送消息的方式有关，如果使用同步方式send（msg）发送，在DefaultMQProducer内部有个自动重试逻辑，其中一个Broker停了，会自动向另一个Broker发消息，不会发生丢消息现象。如果使用异步方式发送send（msg，callback），或者用sendOneWay方式，会丢失切换过程中的消息。因为在异步和sendOneWay这两种发送方式下，Producer.setRetryTimesWhenSendFailed设置不起作用，发送失败不会重试。DefaultMQProducer默认每30秒到NameServer请求最新的路由消息，Producer如果获取不到已停止的Broker下的队列信息，后续就自动不再向这些队列发送消息。

如果Producer程序能够暂停，在有一个Master和一个Slave的情况下也可以顺利切换。可以关闭Producer后关闭Master Broker，这个时候所有的读取都会被定向到Slave机器，消费消息不受影响。把Master Broker机器置换完后，基于原来的数据启动这个Master Broker，然后再启动Producer程序正常发送消息。

用Linux的kill pid命令就可以正确地关闭Broker，BrokerController下有个shutdown函数，这个函数被加到了ShutdownHook里，当用Linux的kill命令时（不能用kill-9），shutdown函数会先被执行。也可以通过RocketMQ提供的工具（mqshutdown broker）来关闭Broker，它们的原理是一样的。

6.4 各种故障对消息的影响

我们期望消息队列集群一直可靠稳定地运行，但有时候故障是难免的，本节我们列出可能的故障情况，看看如何处理：

1）Broker正常关闭，启动；

2）Broker异常Crash，然后启动；

3）OS Crash，重启；

4）机器断电，但能马上恢复供电；

5）磁盘损坏；

6）CPU、主板、内存等关键设备损坏。

假设现有的RocketMQ集群，每个Topic都配有多Master角色的Broker供写入，并且每个Master都至少有一个Slave机器（用两台物理机就可以实现上述配置），我们来看看在上述情况下消息的可靠性情况。

第1种情况属于可控的软件问题，内存中的数据不会丢失。如果重启过程中有持续运行的Consumer，Master机器出故障后，Consumer会自动重连到对应的Slave机器，不会有消息丢失和偏差。当Master角色的机器重启以后，Consumer又会重新连接到Master机器（注意在启动Master机器的时候，如果Consumer正在从Slave消费消息，不要停止Consumer。假如此时先停止Consumer后再启动Master机器，然后再启动Consumer，这个时候Consumer就会去读Master机器上已经滞后的offset值，造成消息大量重复）。

如果第1种情况出现时有持续运行的Producer，一台Master出故障后，Producer只能向Topic下其他的Master机器发送消息，如果Producer采用同步发送方式，不会有消息丢失。

第2、3、4种情况属于软件故障，内存的数据可能丢失，所以刷盘策略不同，造成的影响也不同，如果Master、Slave都配置成

SYNC_FLUSH，可以达到和第1种情况相同的效果。

第5、6种情况属于硬件故障，发生第5、6种情况的故障，原有机器的磁盘数据可能会丢失。如果Master和Slave机器间配置成同步复制方式，某一台机器发生5或6的故障，也可以达到消息不丢失的效果。如果Master和Slave机器间是异步复制，两次Sync间的消息会丢失。

总的来说，当设置成：

1）多Master，每个Master带有Slave；

2）主从之间设置成SYNC_MASTER；

3）Producer用同步方式写；

4）刷盘策略设置成SYNC_FLUSH。

就可以消除单点依赖，即使某台机器出现极端故障也不会丢消息。

6.5 消息优先级

有些场景，需要应用程序处理几种类型的消息，不同消息的优先级不同。RocketMQ是个先入先出的队列，不支持消息级别或者Topic级别的优先级。业务中简单的优先级需求，可以通过间接的方式解决，下面列举三种优先级相关需求的具体处理方法。

第一种是比较简单的情况，如果当前Topic里有多种相似类型的消息，比如类型AA、AB、AC，当AB、AC的消息量很大，但是处理速度比较慢的时候，队列里会有很多AB、AC类型的消息在等候处理，这个时候如果有少量AA类型的消息加入，就会排在AB、AC类型消息后面，需要等候很长时间才能被处理。

如果业务需要AA类型的消息被及时处理，可以把这三种相似类型的消息分拆到两个Topic里，比如AA类型的消息在一个单独的Topic，AB、AC类型的消息在另外一个Topic。把消息分到两个Topic中以后，应用程序创建两个Consumer，分别订阅不同的Topic，这样消息AA在单独的Topic里，不会因为AB、AC类型的消息太多而被长时间延时处理。

第二种情况和第一种情况类似，但是不用创建大量的Topic。举个实际应用场景：一个订单处理系统，接收从100家快递门店过来的请求，把这些请求通过Producer写入RocketMQ；订单处理程序通过Consumer从队列里读取消息并处理，每天最多处理1万单。如果这100个快递门店中某几个门店订单量大增，比如门店一接了个大客户，一个上午就发出2万单消息请求，这样其他的99家门店可能被迫等待门店一的2万单处理完，也就是两天后订单才能被处理，显然很不公平。

这时可以创建一个Topic，设置Topic的MessageQueue数量超过100个，Producer根据订单的门店号，把每个门店的订单写入一个MessageQueue。DefaultMQPushConsumer默认是采用循环的方式逐个读取一个Topic的所有MessageQueue，这样如果某家门店订单量大增，这家门店对应的MessageQueue消息数增多，等待时间增长，但不会造成其他家门店等待时间增长。

DefaultMQPushConsumer默认的pullBatchSize是32，也就是每次从某个MessageQueue读取消息的时候，最多可以读32个。在上面的场景

中，为了更加公平，可以把pullBatchSize设置成1。

第三种情况是强优先级需求，上两种情况对消息的“优先级”要求不高，更像一个保证公平处理的机制，避免某类消息的增多阻塞其他类型的消息。现在有一个应用程序同时处理TypeA、TypeB、TypeC三类消息。TypeA处于第一优先级，要确保只要有TypeA消息，必须优先处理；TypeB处于第二优先级；TypeC处于第三优先级。对这种要求，或者逻辑更复杂的要求，就要用户自己编码实现优先级控制，如果上述的三类消息在一个Topic里，可以使用PullConsumer，自主控制MessageQueue的遍历，以及消息的读取；如果上述三类消息在三个Topic下，需要启动三个Consumer，实现逻辑控制三个Consumer的消费。

6.6 本章小结

本章根据使用场景，讨论如何“可靠”地收发消息。即在要求消息顺序的场景下，如何既能并发执行，又能保证消息顺序；然后分析在可能的故障场景下，如何应对以保证不丢消息、不中断服务。RocketMQ在设计上，有重试机制来保证消息不丢，造成的结果是可能存在消息重复，这一点需要用户根据具体业务场景来处理。下一章将讨论处理大数据量消息的方法。

第7章　吞吐量优先的使用场景

本章介绍在大流量场景下，提高RocketMQ集群吞吐量的一些方法，有些方法当服务器出异常时会增大丢消息的概率，用户需要根据业务需求酌情使用。

7.1 在Broker端进行消息过滤

在Broker端进行消息过滤，可以减少无效消息发送到Consumer，少占用网络带宽从而提高吞吐量。Broker端有三种方式进行消息过滤。

7.1.1 消息的Tag和Key

对一个应用来说，尽可能只用一个Topic，不同的消息子类型用Tag来标识（每条消息只能有一个Tag），服务器端基于Tag进行过滤，并不需要读取消息体的内容，所以效率很高。发送消息设置了Tag以后，消费方在订阅消息时，才可以利用Tag在Broker端做消息过滤。

其次是消息的Key。对发送的消息设置好Key，以后可以根据这个Key来查找消息。所以这个Key一般用消息在业务层面的唯一标识码来表示，这样后续查询消息异常，消息丢失等都很方便。Broker会创建专门的索引文件，来存储Key到消息的映射，由于是哈希索引，应尽量使Key唯一，避免潜在的哈希冲突。

Tag和Key的主要差别是使用场景不同，Tag用在Consumer的代码中，用来进行服务端消息过滤，Key主要用于通过命令行查询消息。

7.1.2 通过Tag进行过滤

用Tag方式进行过滤的方法是传入感兴趣的Tag标签，Tag标签是一个普通字符串，是在创建Message的时候添加的，一个Message只能有一个Tag。使用Tag方式过滤非常高效，Broker端可以在ConsumeQueue中做这种过滤，只从CommitLog里读取过滤后被命中的消息。看一下ConsumerQueue的存储格式，如图7-1所示。

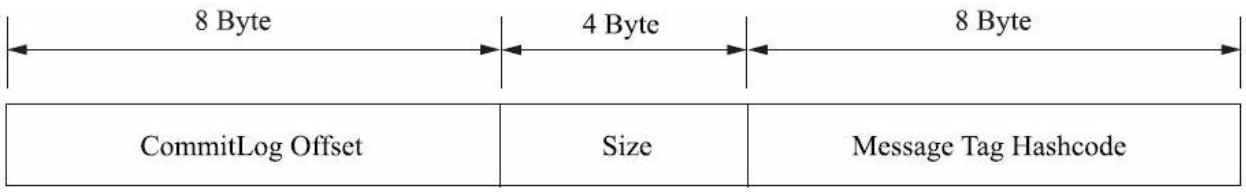


图7-1 ConsumerQueue的存储格式

Consume Queue的第三部分存储的是Tag对应的hashcode，是一个定长的字符串，通过Tag过滤的过程就是对比定长的hashcode。经过hashcode对比，符合要求的消息被从CommitLog读取出来，不用担心Hash冲突问题，消息在被消费前，会对比完整的Message Tag字符串，消除Hash冲突造成的误读。

7.1.3　用SQL表达式的方式进行过滤

使用Tag方式过滤虽然高效，但是支持的逻辑比较简单，在构造Message的时候，还可以通过putUserProperty函数来增加多个自定义的属性，基于这些属性可以做复杂的过滤逻辑，如代码清单7-1所示。

代码清单7-1　在消息中增加自定义属性

```
Message msg = new Message("TopicTest",
    tag,
    ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET)
);
// Set some properties.
msg.putUserProperty("a", String.valueOf(i));
msg.putUserProperty("b",  “hello”);
```

代码中这个消息就有了两个特殊的属性值a和b，我们用类似SQL表达式的方式对消息进行过滤，用法如下（目前只支持在PushConsumer中实现这种过滤）：

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_gro
consumer.registerMessageListener(new MessageListenerConcurrently()
{     @Override     public ConsumeConcurrentlyStatus consumeMessage
    (List<MessageExt> msgs, ConsumeConcurrentlyContext context)
{         return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;     } }); consumer.start
```

类似SQL的过滤表达式，支持如下语法：

·数字对比，比如>、>=、<、<=、BETWEEN、=；

·字符串对比，比如=、<>、IN；

·IS NULL or IS NOT NULL；

·逻辑符号AND、OR、NOT。

支持的数据类型：

·数字型，比如123、3.1415；

·字符型，比如'abc'、注意必须用单引号；

·NULL，这个特殊字符；

·布尔型，TRUEorFALSE。

SQL表达式方式的过滤需要Broker先读出消息里的属性内容，然后做SQL计算，增大磁盘压力，没有Tag方式高效。

7.1.4 Filter Server方式过滤

Filter Server是一种比SQL表达式更灵活的过滤方式，允许用户自定义Java函数，根据Java函数的逻辑对消息进行过滤。

要使用Filter Server，首先要在启动Broker前在配置文件里加上filterServer-Nums=3这样的配置，Broker在启动的时候，就会在本机启动3个Filter Server进程。Filter Server类似一个RocketMQ的Consumer进程，它从本机Broker获取消息，然后根据用户上传过来的Java函数进行过滤，过滤后的消息再传给远端的Consumer。这种方式会占用很多Broker机器的CPU资源，要根据实际情况谨慎使用。上传的java代码也要经过检查，不能有申请大内存、创建线程等这样的操作，否则容易造成Broker服务器宕机。实现过滤逻辑的示例如代码清单7-2所示。

代码清单7-2　实现过滤逻辑的代码示例

```
public class MessageFilterImpl implements MessageFilter {
    @Override
    public boolean match(MessageExt msg) {
        String property = msg.getUserProperty("SequenceId");
        if (property != null) {
            int id = Integer.parseInt(property);
            if ((id % 3) == 0 && (id > 10)) {
                return true;
            }
        }
        return false;
    }
}
```

上面代码实现了过滤逻辑，它是根据消息的“SequenceId”这个属性来过滤的，其实不一定要根据消息属性来过滤，也可以根据消息体的内容或其他特征过滤，如代码清单7-3所示。

代码清单7-3　使用FilterServer的Consumer示例

```
public static void main(String[] args) throws InterruptedException, MQClientExceptio
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("Consumer-GroupNamecc
    // 使用Java代码，在服务器做消息过滤
    String filterCode = MixAll.file2String("/home/admin/MessageFilterImpl.java");
        consumer.subscribe("TopicFilter7", "com.alibaba.rocketmq.example.filter.Mess
        consumer.registerMessageListener(new MessageListenerConcurrently() {
```

```
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
                ConsumeConcurrentlyContext context) {
            System.out.println(Thread.currentThread().getName() + " Receive New Mess
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        consumer.start();
        System.out.println("Consumer Started.");
    }
```

在使用Filter Server的Consumer例子中，主要是把实现过滤逻辑的类作为参数传到Broker端，Broker端的Filter Server会解析这个类，然后根据match函数里的逻辑进行过滤。

7.2 提高Consumer处理能力

当Consumer的处理速度跟不上消息的产生速度，会造成越来越多的消息积压，这个时候首先查看消费逻辑本身有没有优化空间，除此之外还有三种方法可以提高Consumer的处理能力。

（1）提高消费并行度

在同一个ConsumerGroup下（Clustering方式），可以通过增加Consumer实例的数量来提高并行度，通过加机器，或者在已有机器中启动多个Consumer进程都可以增加Consumer实例数。注意总的Consumer数量不要超过Topic下Read Queue数量，超过的Consumer实例接收不到消息。此外，通过提高单个Consumer实例中的并行处理的线程数，可以在同一个Consumer内增加并行度来提高吞吐量（设置方法是修改consumeThreadMin和consumeThreadMax）。

（2）以批量方式进行消费

某些业务场景下，多条消息同时处理的时间会大大小于逐个处理的时间总和，比如消费消息中涉及update某个数据库，一次update10条的时间会大大小于十次update1条数据的时间。这时可以通过批量方式消费来提高消费的吞吐量。实现方法是设置Consumer的consumeMessageBatchMaxSize这个参数，默认是1，如果设置为N，在消息多的时候每次收到的是个长度为N的消息链表。

（3）检测延时情况，跳过非重要消息

Consumer在消费的过程中，如果发现由于某种原因发生严重的消息堆积，短时间无法消除堆积，这个时候可以选择丢弃不重要的消息，使Consumer尽快追上Producer的进度，如代码清单7-4所示。

代码清单7-4　判断消息堆积并处理示例

```
public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcur
long Offset = msgs.get(0).getQueueOffset();
String maxOffset = msgs.get(0).getProperty(Message.PROPERTY_MAX_OFFSET);     long di
if (diff > 90000) {
```

```
return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
}
//正常消费消息
 return ConsumeConcurrentlyStatus.CONSUME_SUCCESS; }
```

如代码所示，当某个队列的消息数堆积到90000条以上，就直接丢弃，以便快速追上发送消息的进度。

7.3 Consumer的负载均衡

上一节中讲到，想要提高Consumer的处理速度，可以启动多个Consumer并发处理，这个时候就涉及如何在多个Consumer之间负载均衡的问题，接下来结合源码分析Consumer的负载均衡实现。

要做负载均衡，必须知道一些全局信息，也就是一个ConsumerGroup里到底有多少个Consumer，知道了全局信息，才可以根据某种算法来分配，比如简单地平均分到各个Consumer。在RocketMQ中，负载均衡或者消息分配是在Consumer端代码中完成的，Consumer从Broker处获得全局信息，然后自己做负载均衡，只处理分给自己的那部分消息。

7.3.1 DefaultMQPushConsumer的负载均衡

DefaultMQPushConsumer的负载均衡过程不需要使用者操心，客户端程序会自动处理，每个DefaultMQPushConsumer启动后，会马上会触发一个doRebalance动作；而且在同一个ConsumerGroup里加入新的DefaultMQPush-Consumer时，各个Consumer都会被触发doRebalance动作。

如图7-2所示，具体的负载均衡算法有五种，默认用的是第一种AllocateMessageQueueAveragely。负载均衡的结果与Topic的Message Queue数量，以及ConsumerGroup里的Consumer的数量有关。负载均衡的分配粒度只到Message Queue，把Topic下的所有Message Queue分配到不同的Consumer中，所以Message Queue和Consumer的数量关系，或者整除关系影响负载均衡结果。

图7-2　RocketMQ客户端负载均衡策略

以AllocateMessageQueueAveragely策略为例，如果创建Topic的时候，把Message Queue数设为3，当Consumer数量为2的时候，有一个Consumer需要处理Topic三分之二的消息，另一个处理三分之一的消息；当Consumer数量为4的时候，有一个Consumer无法收到消息，其他

3个Consumer各处理Topic三分之一的消息。可见Message Queue数量设置过小不利于做负载均衡，通常情况下，应把一个Topic的Message Queue数设置为16。

7.3.2 DefaultMQPullConsumer的负载均衡

Pull Consumer可以看到所有的Message Queue，而且从哪个Message Queue读取消息，读消息时的Offset都由使用者控制，使用者可以实现任何特殊方式的负载均衡。

DefaultMQPullConsumer有两个辅助方法可以帮助实现负载均衡，一个是registerMessageQueueListener函数，如代码清单7-5所示。

代码清单7-5　registerMessageQueueListener

```
Consumer.registerMessageQueueListener("TOPICNAME", new MessageQueue-Listener() {
public void MessageQueueChanged(String Topic, Set<MessageQueue> mqAll, Set<MessageQu
```

registerMessageQueueListener函数在有新的Consumer加入或退出时被触发。另一个辅助工具是MQPullConsumerScheduleService类，使用这个Class类似使用DefaultMQPushConsumer，但是它把Pull消息的主动性留给了使用者，如代码清单7-6所示。

代码清单7-6　使用MQPullConsumerScheduleService示例

```
public class PullConsumerServiceTest {
    public static void main(String[] args) throws MQClientException {
        final MQPullConsumerScheduleService scheduleService = new MQPull-ConsumerSch
        scheduleService.getDefaultMQPullConsumer().setNamesrvAddr("localh-ost:9876")
        scheduleService.setMessageModel(MessageModel.CLUSTERING );
        scheduleService.registerPullTaskCallback("testPullConsumer", new PullTaskCal
            public void doPullTask(MessageQueue mq, PullTaskContext context) {
                MQPullConsumer Consumer = context.getPullConsumer();
                try {
                    long Offset = Consumer.fetchConsumeOffset(mq, false);
                    if (Offset < 0)
                        Offset = 0;
                    PullResult pullResult = Consumer.pull(mq, "*", Offset, 32);
                    System.out.printf("%s%n", Offset + "\t" + mq + "\t" + pullResult
                    switch (pullResult.getPullStatus()) {
                        case FOUND:
                            break;
                        case NO_MATCHED_MSG:
                            break;
                        case NO_NEW_MSG:
                        case OFFSET_ILLEGAL:
                            break;
                        default:
                            break;
                    }
```

```
                    Consumer.updateConsumeOffset(mq, pullResult.getNextBeginOffset()
                    context.setPullNextDelayTimeMillis(1000);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
        scheduleService.start();
    }
}
```

然后我们看一看在MQPullConsumerScheduleService类的实现里，实现负载均衡的代码，如代码清单7-7所示。

代码清单7-7　MQPullConsumerScheduleService的负载均衡实现

```
class MessageQueueListenerImpl implements MessageQueueListener {
    @Override
    public void MessageQueueChanged(String Topic, Set<MessageQueue> mqAll, Set<Messa
        MessageModel MessageModel =
            MQPullConsumerScheduleService.this.defaultMQPullConsumer.getMessageModel
        switch (MessageModel) {
            case BROADCASTING:
                MQPullConsumerScheduleService.this.putTask(Topic, mqAll);
                break;
            case CLUSTERING :
                MQPullConsumerScheduleService.this.putTask(Topic, mqDivided);
                break;
            default:
                break;
        }
    }
}
```

从源码中可以看出，用户通过更改MessageQueueListenerImpl的实现来做自己的负载均衡策略。

7.4 提高Producer的发送速度

发送一条消息出去要经过三步，一是客户端发送请求到服务器，二是服务器处理该请求，三是服务器向客户端返回应答，一次消息的发送耗时是上述三个步骤的总和。在一些对速度要求高，但是可靠性要求不高的场景下，比如日志收集类应用，可以采用Oneway方式发送，Oneway方式只发送请求不等待应答，即将数据写入客户端的Socket缓冲区就返回，不等待对方返回结果，用这种方式发送消息的耗时可以缩短到微秒级。

另一种提高发送速度的方法是增加Producer的并发量，使用多个Producer同时发送，我们不用担心多Producer同时写会降低消息写磁盘的效率，RocketMQ引入了一个并发窗口，在窗口内消息可以并发地写入DirectMem中，然后异步地将连续一段无空洞的数据刷入文件系统当中。顺序写CommitLog可让RocketMQ无论在HDD还是SSD磁盘情况下都能保持较高的写入性能。目前在阿里内部经过调优的服务器上，写入性能达到90万+的TPS，我们可以参考这个数据进行系统优化。

在Linux操作系统层级进行调优，推荐使用EXT4文件系统，IO调度算法使用deadline算法。

如图7-3所示，EXT4创建/删除文件的性能比EXT3及其他文件系统要好，RocketMQ的CommitLog会有频繁的创建/删除动作。

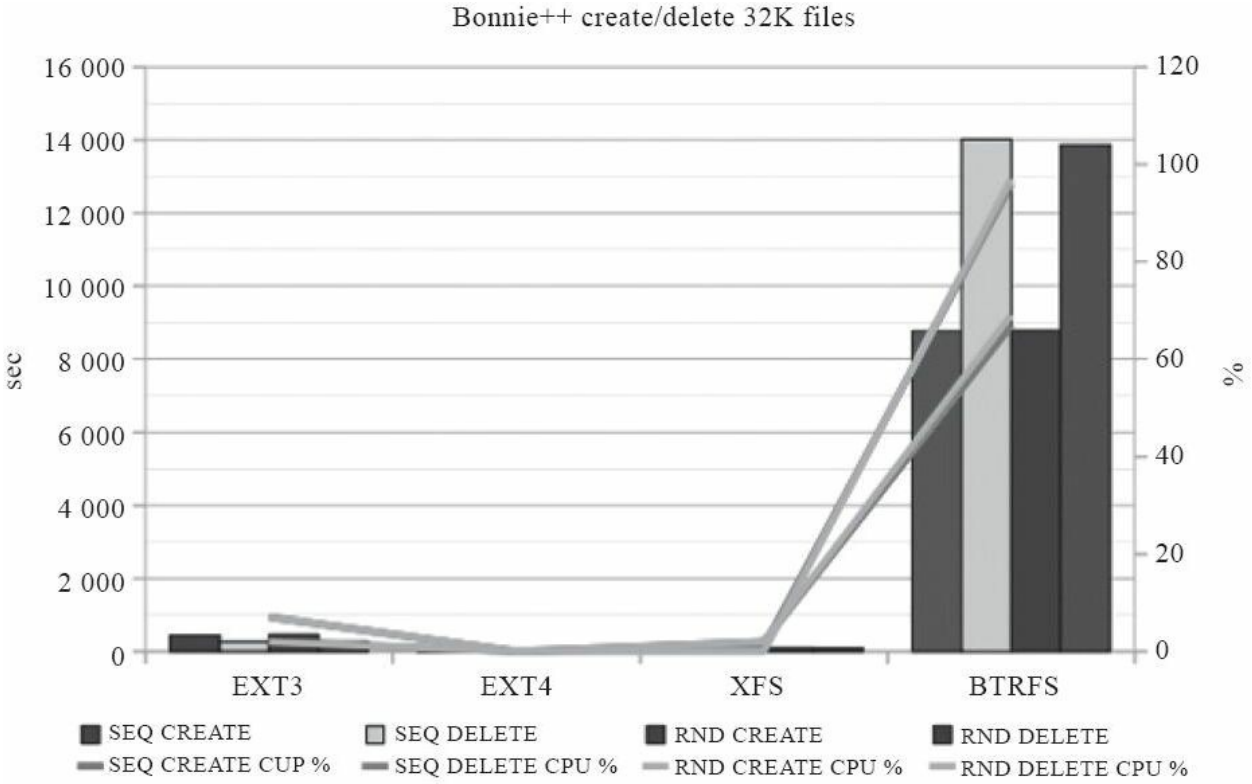


图7-3　几种文件系统在Bonnie++中创建/删除32K文件需要的时间

另外，IO调度算法也推荐调整为deadline。deadline算法大致思想如下：实现四个队列，其中两个处理正常的read和write操作，另外两个处理超时的read和write操作。正常的read和write队列中，元素按扇区号排序，进行正常的IO合并处理以提高吞吐量。因为IO请求可能会集中在某些磁盘位置，这样会导致新来的请求一直被合并，可能会有其他磁盘位置的IO请求被饿死。超时的read和write的队列中，元素按请求创建时间排序，如果有超时的请求出现，就放进这两个队列，调度算法保证超时（达到最终期限时间）的队列中的IO请求会优先被处理。

7.5 系统性能调优的一般流程

这里讨论的系统是指能完成某项功能的软硬件整体，比如我们用RocketMQ，加上自己写的Producer、Consumer程序，部署到一台服务器上，组成一个消息处理系统。本节介绍对这类系统进行调优的基本流程，供读者参考。

首先是搭建测试环境，查看硬件利用率。把测试系统搭建好以后，要想办法模拟实际使用时的情况，并且逐步增大请求量，同时检测系统的TPS。在请求量增大到一定程度时，系统的QPS达到峰值，这个时候维持这种请求量，保持系统在峰值状态下运行。然后查看此时系统的硬件使用情况：

（1）使用TOP命令查看CPU和内存的利用率

```
Tasks: 109 total,   1 running, 108 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.2 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  8010440 total,  1556880 free,  1626048 used,  4827512 buff/cache
KiB Swap:   0 total,  0 free,  0 used.  6058356 avail Mem
```

上面的数据显示，CPU有99.8%空闲；内存总共8G，有大约1.5G空闲。

（2）使用Linux的sar命令查看网卡使用情况

```
#sar -n DEV 2 10
Average: IFACE rxpck/s txpck/s rxkB/s txkB/s rxcmp/s txcmp/s rxmcst/s
Average: eth0   6.03  6.18   1.39 0.99   0.00   0.00   0.00
Average: eth1   4.41   3.82   0.42   0.98   0.00   0.00   0.00
```

·IFACE：LAN接口，网络设备的名称。

·rxpck/s：每秒钟接收的数据包。

·txpck/s：每秒钟发送的数据包。

·rxbyt/s：每秒钟接收的字节数。

·txbyt/s：每秒钟发送的字节数。

·rxcmp/s：每秒钟接收的压缩数据包。

·txcmp/s：每秒钟发送的压缩数据包。

·rxmcst/s：每秒钟接收的多播数据包。

如果想进一步验证网卡是否达到了极限值，可以使用iperf3命令查看。还可以用netstat–t查看网卡的连接情况，看是否有大量连接造成堵塞。

然后用iostat查看磁盘的使用情况：

```
#iostat -xdm 1
Linux 3.10.0-514.6.1.el7.x86_64 (iZ2zehfpu32ir7r3vlhhuwZ)   12/28/2017
    _x86_64_(4 CPU)

Device:   rrqm/s   wrqm/s  r/s  w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_av
vda   0.00  1.04    0.01    1.15  0.00  0.01    19.84  0.00    2.45
    1.58    2.46   0.39   0.05
vdb   0.00  0.00    0.00    0.00  0.00  0.00    14.75  0.00    0.11
    0.11    0.00   0.09   0.00
```

经过上面的一系列检查，应该能够找到系统的瓶颈。比如瓶颈是在CPU、网卡还是磁盘？可以先确定网卡和磁盘是否繁忙，这两个中如果有一个被占满了，问题就可以被直接定位了。比如网卡打满了，我们可以判断是发送的数据量超出了网卡的带宽，可以考虑更换高速网卡，或者更新程序减少数据发送量。

还有一种情况是这三者都没有到使用极限，这也是一种比较常见而且有优化空间的情况，这种情况说明CPU利用率没有发挥出来，比如可能是锁的机制有bug，造成线程阻塞。

对于Java程序来说，接下来可以用Java的profiling工具来找出程序的具体问题，比如jvisualvm、jstack、perfJ等。

通过上面这些工具，可以逐步定位出是哪些Java线程比较慢，哪个函数占用的时间多，是否因为存在锁造成了忙等的情况，然后通过不断的更改测试，找到影响性能的关键代码，最终解决问题。

7.6 本章小结

本章重点关注性能，关注在大消息量的情况下，如何提高RocketMQ的吞吐量。首先介绍了消息过滤，在服务端进行消息过滤可以减少无效消息传输造成的带宽浪费，Tag是最常用的一种高效过滤方式，此外还可以用SQL表达式、FilterServer来过滤消息。

另一个提高吞吐量的方法是增加集群的机器数量，提高并发性，要根据实际场景增加Broker、Consumer或Producer角色的机器数量。

第8章　和其他系统交互

8.1　在SpringBoot中使用RocketMQ

Spring Boot因为方便易用，在Java开发者中大受好评，被誉为“Spring的第二春”，本章将说明如何在Spring项目中快速使用RocketMQ。

8.1.1 直接使用

在Spring Boot项目中，使用某个新的组件第一步通常是加入这个组件的依赖。下面以Maven为例，说明如何在pom.xml中加入RocketMQ的依赖，如代码清单8-1所示。

代码清单8-1 Maven方式的RocketMQ依赖

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.2.0</version>
</dependency>
```

有了这个依赖，就可以在Spring Boot项目中开发RocketMQ的Producer和Consumer程序了。

使用RocketMQ集群，有很多参数要设置，我们可以在application.properties文件里加入自己命名的参数，然后通过@Value注解引入。几个重要的参数是：NameServer的地址、GoupName名称和Topic名称。此外还有一些针对Producer或Consumer的参数，可以写到properties文件里，也可以写到程序里。

依赖配置都做好以后，就可以着手开发Producer和Consumer程序了。我们可以把发送消息和消费消息的功能封装成Service，供其他代码引用。Producer和Consumer的初始化比较慢，不建议每发一个消息或者消费一个消息就启动和注销对应的Object，所以适合把初始化操作代码写到@PostConstruct函数里，把关闭操作代码写到@PreDestroy函数里。Spring Boot项目中的Producer程序示例如代码清单8-2所示。

代码清单8-2 Spring Boot项目中的Producer服务

```
@Service
public class ProducerService {
private DefaultMQProducer producer = null;
    @PostConstruct
    public void initMQProducer() {
        producer = new DefaultMQProducer(“producerGoupName”);
        producer.setNamesrvAddr(metaqNameserver);
        producer.setRetryTimesWhenSendFailed(3);
```

```
            try {
                producer.start();
            } catch (MQClientException e) {
                e.printStackTrace();
            }
        }
        public void send(String topic, String msg) {
            Message msg = new Message(topic, "", "", msg.getBytes());

            try {
                producer.send(msg);
                return;
            } catch (Exception e) {
                e.printStackTrace();
            }
                return;
        }
        @PreDestroy
        public void shutDownProducer() {
            if (producer != null) {
                producer.shutdown();
            }
        }
    }
```

使用Consumer的方式和使用Producer类似，但是具体设置会因为使用的具体Class不同而不同。调用shutdown函数是必要的，否则可能因为程序被强制关闭而丢消息。

8.1.2 通过Spring Messaging方式使用

直接使用的方式比较简单，也足够灵活，但不是很“Spring Style”，Spring Boot对于消息传递，有统一的接口模板，基于这个模板可以对接各种类型的消息通信组件，比如Kafka、RabbitMQ、RocketMQ等。使用这种方式，其基于不同消息队列收发消息的代码类似，方便在不同的消息队列间切换。

具体使用流程分为三个步骤：添加依赖、配置参数和引入模板。添加RocketMQ插件示例，如代码清单8-3所示。

代码清单8-3　Spring Boot的RocketMQ插件

```
<!--在pom.xml中添加依赖-->
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>spring-boot-starter-rocketmq</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>
```

如果mvn找不到这个依赖，可以在GitHub上下载源码，本地构建。

然后是在properties文件中加入配置选项，如代码清单8-4所示。

代码清单8-4　Spring Boot的RocketMQ相关配置选项

```
## application.properties
spring.rocketmq.name-server=127.0.0.1:9876
spring.rocketmq.producer.group=my-group
spring.rocketmq.producer.retry-times-when-send-async-failed=0
spring.rocketmq.producer.send-msg-timeout=300000
spring.rocketmq.producer.compress-msg-body-over-howmuch=4096
spring.rocketmq.producer.max-message-size=4194304
spring.rocketmq.producer.retry-another-broker-when-not-store-ok=false
spring.rocketmq.producer.retry-times-when-send-failed=2
```

更多的配置选项，可以到源码中查找。由于Spring Boot项目和RocketMQ项目变化很快，具体如何以Spring Messaging的方式发送和接收消息，大家可以自行搜索相关的示例和说明。最新的文档可以参考Spring Boot文档的Messaging部分，以及GitHub中的rocketmq-externals项

目。

8.2 直接使用云上RocketMQ

阿里云的很多产品都是来自于集团内部开发的优秀中间件，RocketMQ就是其中之一，阿里云的MQ产品就是基于RocketMQ实现的，后台技术团队同样是开发RocketMQ的团队。

现在产品迭代的节奏越来越快，尤其对于中小型公司来说，直接使用云产品可以省去部署、运维的繁琐工作，加快自身核心产品的上线速度。当业务量上升到一定规模，业务形态基本稳定后，再自己部署、运维或二次开发独立的中间件产品。

如果仔细阅读了前面的章节，参考阿里云MQ的说明文档进行开发就非常容易了，比如阿里云MQ文档中的发送消息Demo，如代码清单8-5所示。

代码清单8-5　阿里云MQ产品发送消息示例

```
public class ProducerTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // 您在 MQ 控制台创建的 Producer ID
        properties.put(PropertyKeyConst.ProducerId, "XXX");
        // 鉴权用 AccessKey，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.AccessKey,"XXX");
        // 鉴权用 SecretKey，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // 设置 TCP 接入域名（此处以公共云的公网接入为例）
        properties.put(PropertyKeyConst.ONSAddr,
          "http://onsaddr-internet.aliyun.com/rocketmq/nsaddr4client-internet");
        Producer producer = ONSFactory.createProducer(properties);
        // 在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可
        producer.start();
        //循环发送消息
        while(true){
            Message msg = new Message( //
                // 在控制台创建的 Topic，即该消息所属的 Topic 名称
                "TopicTestMQ",
                // Message Tag,
                // 可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定
                   过滤条件在 MQ 服务器过滤
                "TagA",
                // Message Body
                // 任何二进制形式的数据， MQ 不做任何干预，
                // 需要 Producer 与 Consumer 协商好一致的序列化和反序列化方式
                "Hello MQ".getBytes());
            // 设置代表消息的业务关键属性，请尽可能全局唯一，以方便您在无法正常收到
               消息情况下，可通过 MQ 控制台查询消息并补发
            // 注意：不设置也不会影响消息正常收发
            msg.setKey("ORDERID_100");
```

```
            // 发送消息，只要不抛异常就是成功
            // 打印 Message ID，以便用于消息发送状态查询
            SendResult sendResult = producer.send(msg);
            System.out.println("Send Message success. Message ID is: " + sendResult.
        }
        // 在应用退出前，可以销毁 Producer 对象
        // 注意：如果不销毁也没有问题
        producer.shutdown();
    }
}
```

这个示例程序和之前介绍的Producer程序比起来，只是把设置GroupName、NameServer地址的部分，换成了阿里云账号的Key，Secret和相应域名，其他部分非常相似。详细信息和最新的文档请参考阿里云MQ产品页面（https://cn.aliyun.com/product/ons ）。

图8-1　阿里云RocketMQ产品页面

8.3 RocketMQ与Spark、Flink对接

Spark和Flink都有对流式计算的支持，如果不考虑并发性的话，可以在自己的程序中启动RocketMQ的Consumer或Producer，负责从RocketMQ集群获取或发送消息，同时再启动Spark或Flink的Client程序，负责和Spark或Flink交互。

如果需要利用Spark、Flink本身的并发处理，需要实现相应的Connector，RocketMQ和Spark的Connector有了实现，代码在https://github.com/apache/rocketmq-externals/tree/master/rocketmq-spark。RocketMQ和Flink的Connector当前正在开发中，有兴趣的也可以参与贡献代码。

8.4 自定义开发运维工具

生产环境的RocketMQ集群，需要持续运行并且要有较高的稳定性，运维是件重要但有时候很繁琐的事，本节介绍运维工具的相关内容。

8.4.1 开源版本运维工具功能介绍

第1章介绍过如何启动运维页面，运维页面打开后，从左至右有7个Tab，分别是：配置、驾驶舱、集群信息、Topic信息、Consumer信息、Producer信息和消息查询，如图8-2所示。

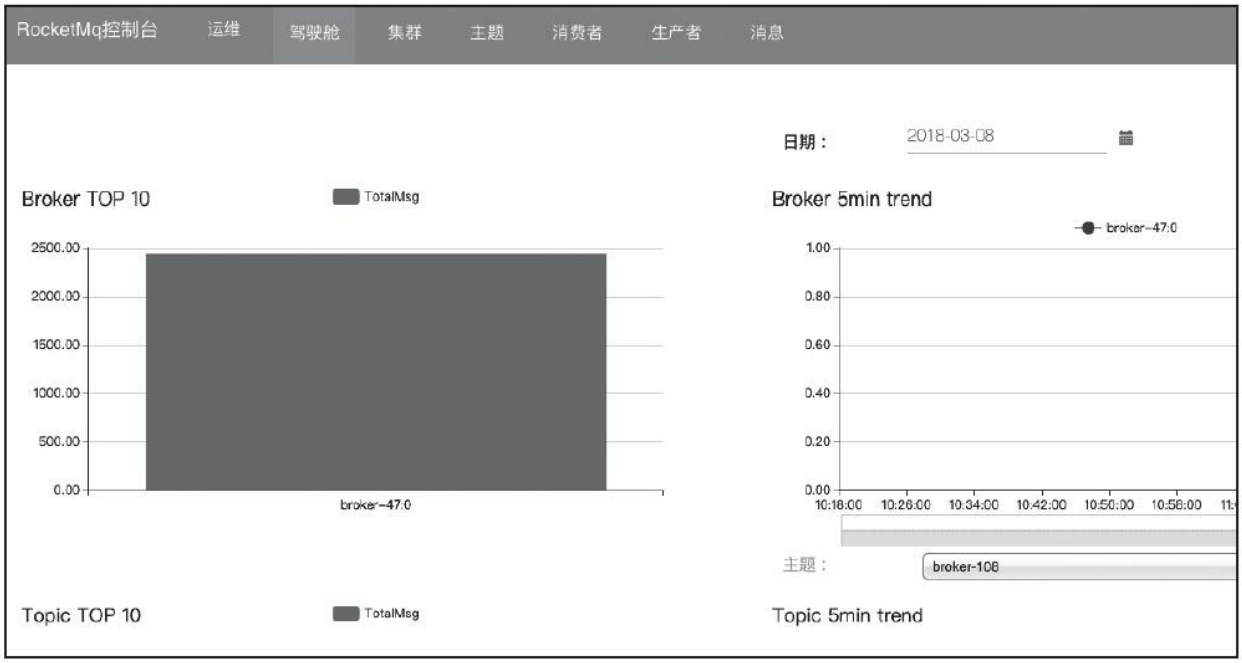


图8-2 RocketMQ控制台

首先在配置页面，设置好NaveServer的地址。修改这个服务是否使用VIPChannel，取决于你的RocketMQ版本，如果版本小于3.5.8，请设置不使用，否则保持默认值（VIPChannel用于实现读写分离，是3.5.8以后的版本才增加的功能）。

在驾驶舱中可以查看Broker的消息量（总量/5分钟图），还可以查看单一主题的消息量（总量/趋势图）。

在集群信息页面，可以查看集群数量、地址、主从的分布情况，还可以查看Broker的运行状态信息和配置信息。

Topic页面展示所有的主题，可以通过搜索框进行过滤，筛选普通/重试/死信类型的主题；还可以添加/更新主题，修改主题的配置参数。

每个参数的含义和MQAdmin命令中updateTopc命令的参数对应。还可以查看每个主题的消息投递状态，消息的路由信息（这个主题的消息会发往哪些Broker，对应Broker的Message Queue信息）。还可以向某个主题发送测试消息和重置消费位点（Offset）。

Consumer信息页面展示所有的消费组，还可以通过搜索框进行搜索，手动刷新页面或每隔五秒定时刷新页面，按照订阅组/数量/TPS/延迟进行排序，添加/更新消费组等。

Producer信息页面，可以通过Topic和Group查询在线的消息生产者信息，信息包含客户端的主机、版本等。

消息查询页面，可以根据Topic的时间、Key和消息ID进行消息查询。消息详情可以展示这条消息的详细内容。消息详情可以查看消息对应的具体消费组的消费情况（如果异常，可以查看具体的异常信息）。可以向指定的消费组重发消息。

8.4.2 基于Tools模块开发自定义运维工具

RocketMQ-Console是一个基于Spring Boot开发的运维页面工具，我们可以参考它的源码进行自定义功能的运维工具开发。

RocketMQ源码中有一个Tools模块，MQAdmin相关命令的实现就在这里，如果我们熟悉了MQAdmin命令的功能，就很容易找到实现某个功能的源码。RocketMQ的Tools模块如图8-3所示。

Tools模块源码中有一个command包，里面列出了各个组件相关的命令，如果想实现自定义的运维功能，可以直接从这里查找并参考它的源码。RocketMQ是使用Java语言开发的，比起Kafka的Scala语言和RabbitMQ的Erlang语言，更容易找到技术人员进行定制开发。大规模使用后，遇到“疑难杂症”也可以直接查看源码，找到深层次的原因。

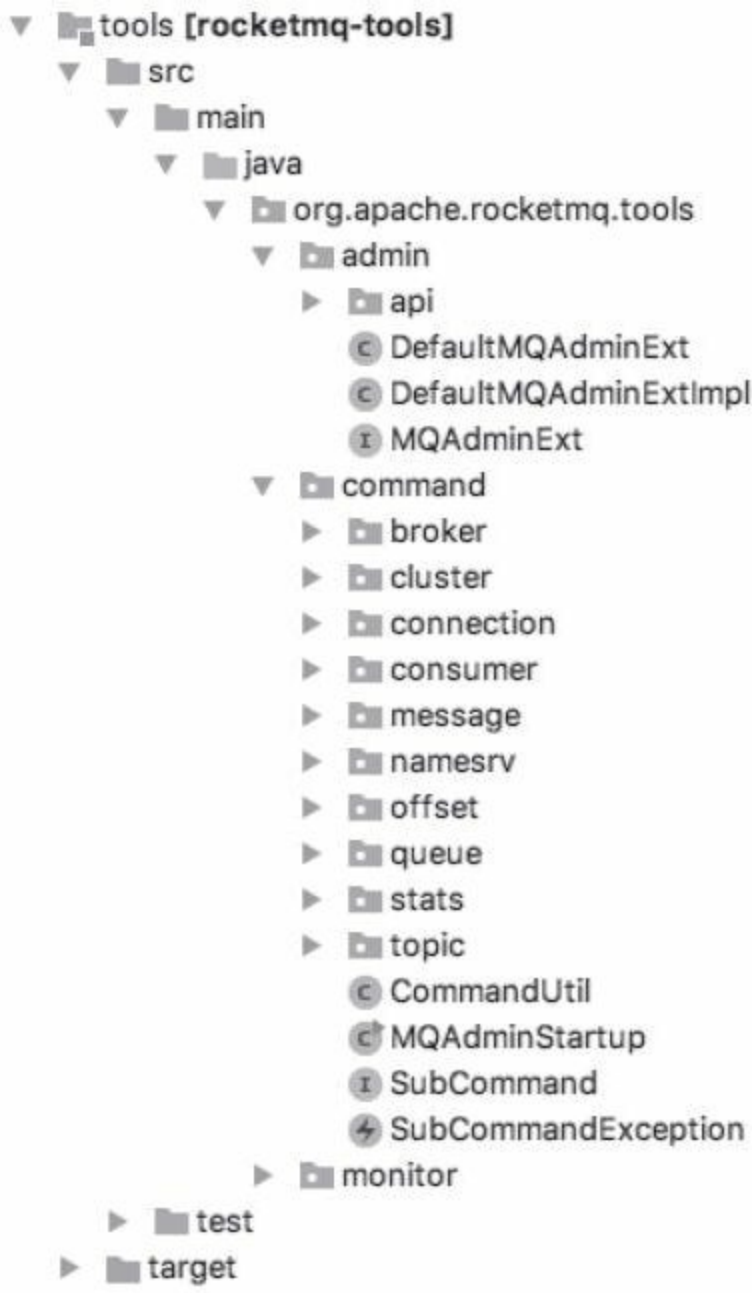


图8-3　RocketMQ的Tools模块

8.5 本章小结

作为一个中间件产品，会比普通软件更多地需要和其他系统打交道，本章介绍了如何与SpringBoot、Spark、Flink等软件进行交互。同时介绍了使用云端的RocketMQ产品，以及自定义开发运维工具的方法。从下一章开始，我们将更深入地介绍RocketMQ，从源码层面进行分析。

第9章　首个Apache中间件顶级项目

本书第1～8章重点介绍的是如何用好RocketMQ，而从本章开始到本书结尾的这些章节，重点介绍RocketMQ的源码。作为一个中间件产品，想真正用好，甚至用来为自己的业务做定制化开发，必须深入了解源码才行。本章介绍RocketMQ项目的概况。

9.1 RocketMQ的前世今生

阿里巴巴消息中间件起源于2001年的五彩石项目，Notify在这期间应运而生，用于交易核心消息的流转。

2010年，B2B开始大规模使用ActiveMQ作为消息内核，随着阿里业务的快速发展，急需一款支持顺序消息，拥有海量消息堆积能力的消息中间件，MetaQ 1.0在2011年诞生。

2012年，MetaQ已经发展到了3.0版本，并抽象出了通用的消息引擎RocketMQ。随后，对RocketMQ进行了开源，阿里的消息中间件正式走入了公众视野。

2015年，RocketMQ已经经历了多年双十一的洗礼，在可用性、可靠性以及稳定性等方面都有出色的表现。与此同时，云计算大行其道，阿里消息中间件基于RocketMQ推出了Aliware MQ 1.0，开始为阿里云上成千上万家企业提供消息服务。

2016年，MetaQ在双十一期间承载了万亿级消息的流转，跨越了一个新的里程碑，同时RocketMQ进入Apache孵化。

图9-1 RocketMQ演进历史

9.2 Apache顶级项目（TLP）之路

RocketMQ的开源模式不是传统意义上的开放内核模式，而是和Apache Hadoop、OpenStack这一类开源平台模式类似，尝试把开源世界和专有世界完美地结合起来，在真正的协作平台上生产专有产品。未来希望像Redhat、CentOS或Fedora这些产品那样，把产品簇的协同发展效应体现在RocketMQ的演进中。

在Apache社区，一个很重要的理念是Community over Code，社区是判断一个孵化项目能否毕业的重要考核标准，有点像我们常说的“客户第一”。除了社区，优秀的代码是必要条件，代码质量不过硬根本不会有一个健康多元的社区，注重代码的同时还要重视社区、设计、产品带给用户的体验。

RocketMQ在进入Apache之前，已经开源了3年时间。历经多次双十一洗礼，RocketMQ在国内积累了一定的口碑，社区也有不错的Active Contributors，但这些还远远不够。在准备申请进入Apache之前，团队甚至包括社区对RocketMQ做了大量重塑工作。如国际化方面，在GitHub上利用sidebar特性重新设计编排了文档，如今已加入了User Guide、Quick Start、Architecture & Design、How to contribute、Community、FAQ这些产品标配的文档结构。代码层面也进行了很多优化，如去除GBK字符，全面拥抱UTF-8，重写API JavaDoc；还有清理代码，优化代码结构；利用JDepend优化组件之间的抽象依赖关系，利用Findbugs扫描代码漏洞，指导规范编码等。交付方面，规范Release流程，使用按New Features、Improvement和Bug分类的Release note。社区层面则开启了全英式互动，发布提问题的技巧。

经过这些准备，RocketMQ完成了从3.0到4.0的悄然升级。而4.0是个过渡版本（和3.0相比，架构层面没有较大的改变），也是在Apache开启孵化的版本。通过孵化，像精细设计、代码Review、编码规约、分支模型、发布规约等软件研发流程被重视起来，无规矩不成方圆，这对一个全球协作的开源项来说尤为重要。

9.3 源码结构

RocketMQ的源码结构如图9-2所示，整个项目是用Maven来管理的，共有十几个模块，主要功能通过broker、client、common、namesrv、remoting、store、tools这几个模块实现。

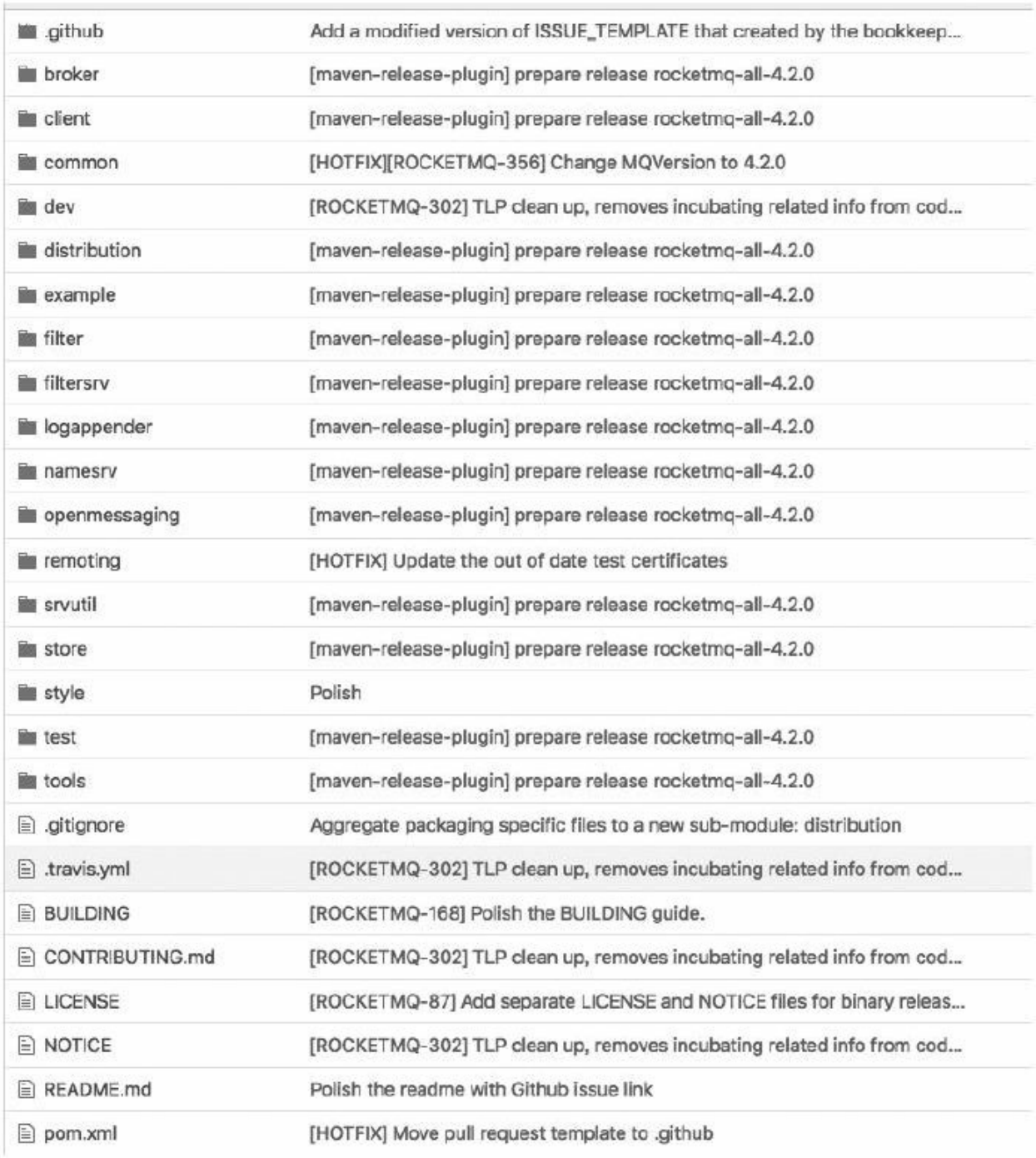
namesrv、broker、client这三个模块前文都有介绍，namesrv是分布式队列集群的协调者，broker实现了消息队列的主体，client包括生产者和消费者，包括使用消息队列的很多辅助方式。common模块包括一些公共的功能类实现，remoting是通信相关功能的实现，store是消息存储的实现，tools主要是管理工具，用来管理集群。

图9-2　RocketMQ源码结构

9.4 不断迭代的代码

RocketMQ的源码在GitHub上一直不断更新，从GitHub上可以下载到最新的代码。RocketMQ在GitHub上的地址是https://github.com/apache/rocketmq，GitHub上的代码结构是未发布版的（见图9-3）。

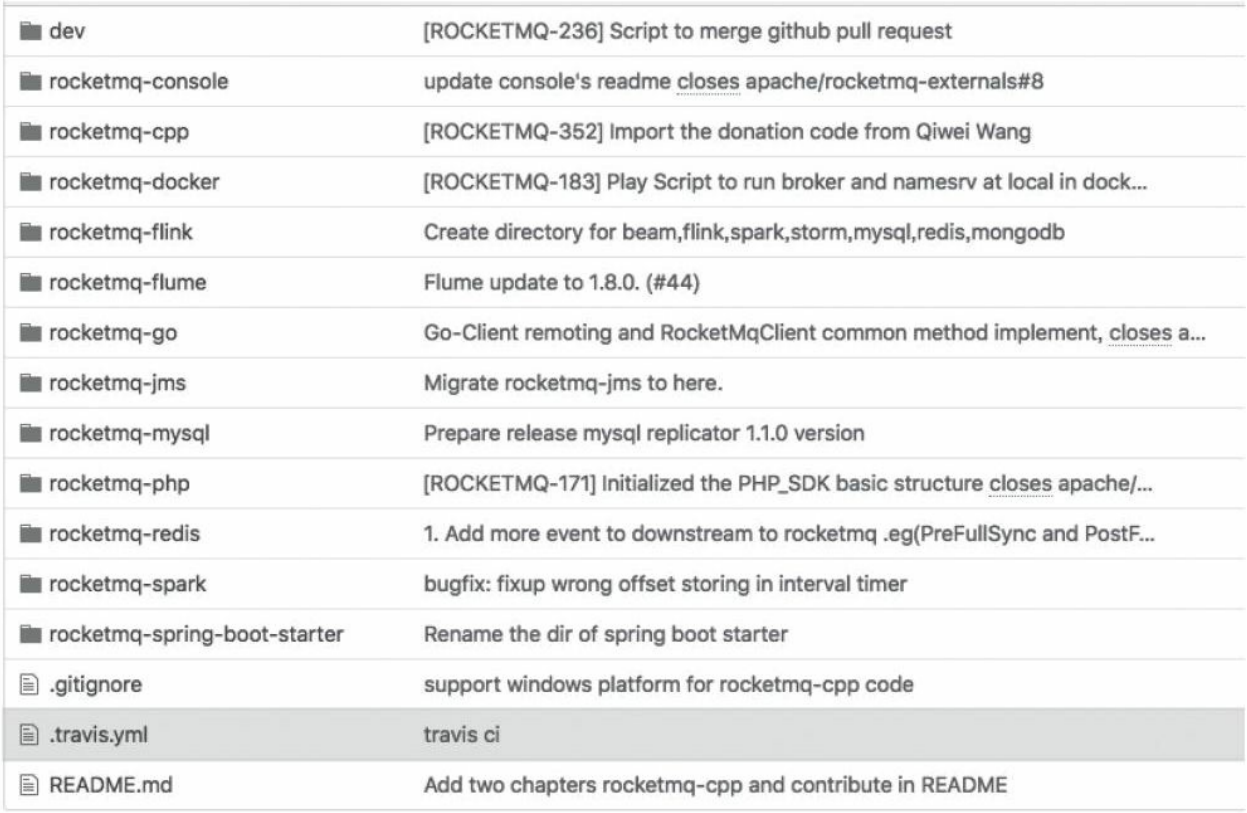
除了RocketMQ主体项目，还有很多和RocketMQ紧密相关的功能，比如管理控制台，以及和Redis、Spark、Flink对接的插件等，这些代码被放到一个单独的GitHub库中，地址是https://github.com/apache/rocketmq-externals 。



.github	Add a modified version of ISSUE_TEMPLATE that created by the bookkeep...
broker	[maven-release-plugin] prepare release rocketmq-all-4.2.0
client	[maven-release-plugin] prepare release rocketmq-all-4.2.0
common	[HOTFIX][ROCKETMQ-356] Change MQVersion to 4.2.0
dev	[ROCKETMQ-302] TLP clean up, removes incubating related info from cod...
distribution	[maven-release-plugin] prepare release rocketmq-all-4.2.0
example	[maven-release-plugin] prepare release rocketmq-all-4.2.0
filter	[maven-release-plugin] prepare release rocketmq-all-4.2.0
filtersrv	[maven-release-plugin] prepare release rocketmq-all-4.2.0
logappender	[maven-release-plugin] prepare release rocketmq-all-4.2.0
namesrv	[maven-release-plugin] prepare release rocketmq-all-4.2.0
openmessaging	[maven-release-plugin] prepare release rocketmq-all-4.2.0
remoting	[HOTFIX] Update the out of date test certificates
srvutil	[maven-release-plugin] prepare release rocketmq-all-4.2.0
store	[maven-release-plugin] prepare release rocketmq-all-4.2.0
style	Polish
test	[maven-release-plugin] prepare release rocketmq-all-4.2.0
tools	[maven-release-plugin] prepare release rocketmq-all-4.2.0
.gitignore	Aggregate packaging specific files to a new sub-module: distribution
.travis.yml	[ROCKETMQ-302] TLP clean up, removes incubating related info from cod...
BUILDING	[ROCKETMQ-168] Polish the BUILDING guide.
CONTRIBUTING.md	[ROCKETMQ-302] TLP clean up, removes incubating related info from cod...
LICENSE	[ROCKETMQ-87] Add separate LICENSE and NOTICE files for binary releas...
NOTICE	[ROCKETMQ-302] TLP clean up, removes incubating related info from cod...
README.md	Polish the readme with Github issue link
pom.xml	[HOTFIX] Move pull request template to .github

图9-3　RocketMQ在GitHub上的代码结构

如果打算贡献代码，官网的指南页面是必读的，地址是http://rocketmq.apache.org/docs/how-to-contribute/，根据文档说明的步骤，提交PR即可。如果不贡献代码，也可以查看某个功能的PR，看看大家的讨论和设计思路，对理解源码也有帮助。



dev	[ROCKETMQ-236] Script to merge github pull request
rocketmq-console	update console's readme closes apache/rocketmq-externals#8
rocketmq-cpp	[ROCKETMQ-352] Import the donation code from Qiwei Wang
rocketmq-docker	[ROCKETMQ-183] Play Script to run broker and namesrv at local in dock...
rocketmq-flink	Create directory for beam,flink,spark,storm,mysql,redis,mongodb
rocketmq-flume	Flume update to 1.8.0. (#44)
rocketmq-go	Go-Client remoting and RocketMqClient common method implement, closes a...
rocketmq-jms	Migrate rocketmq-jms to here.
rocketmq-mysql	Prepare release mysql replicator 1.1.0 version
rocketmq-php	[ROCKETMQ-171] Initialized the PHP_SDK basic structure closes apache/...
rocketmq-redis	1. Add more event to downstream to rocketmq .eg(PreFullSync and PostF...
rocketmq-spark	bugfix: fixup wrong offset storing in interval timer
rocketmq-spring-boot-starter	Rename the dir of spring boot starter
.gitignore	support windows platform for rocketmq-cpp code
.travis.yml	travis ci
README.md	Add two chapters rocketmq-cpp and contribute in README

图9-4　rocket-externals代码结构

9.5 本章小结

RocketMQ是阿里最优秀的中间件之一，本章介绍了RocketMQ的历史，以及其目前作为Apache顶级项目的现状。下一章将从NameServer入手开始分析源码。

第10章　NameServer源码解析

第4章介绍过NameServer的主要功能，功能不多但是很重要，本章分析NameServer的源码，让读者对NameServer有更进一步的了解。

10.1 模块入口代码的功能

本节介绍入口代码的功能，阅读源码的时候，很多人喜欢根据执行逻辑，先从入口代码看起。NameServer部分入口代码主要完成命令行参数解析，初始化Controller的功能。

10.1.1 入口函数

首先看一下NameServer的源码目录（见图10-1）。

NamesrvStartup是模块的启动入口，NamesrvController是用来协块各个调模功能的代码。

我们从启动代码开始分析，找到NamesrvStartup.java里的main函数public static void main（String[]args）{main0（args）；}，发现它又把逻辑转到main0这个函数里。

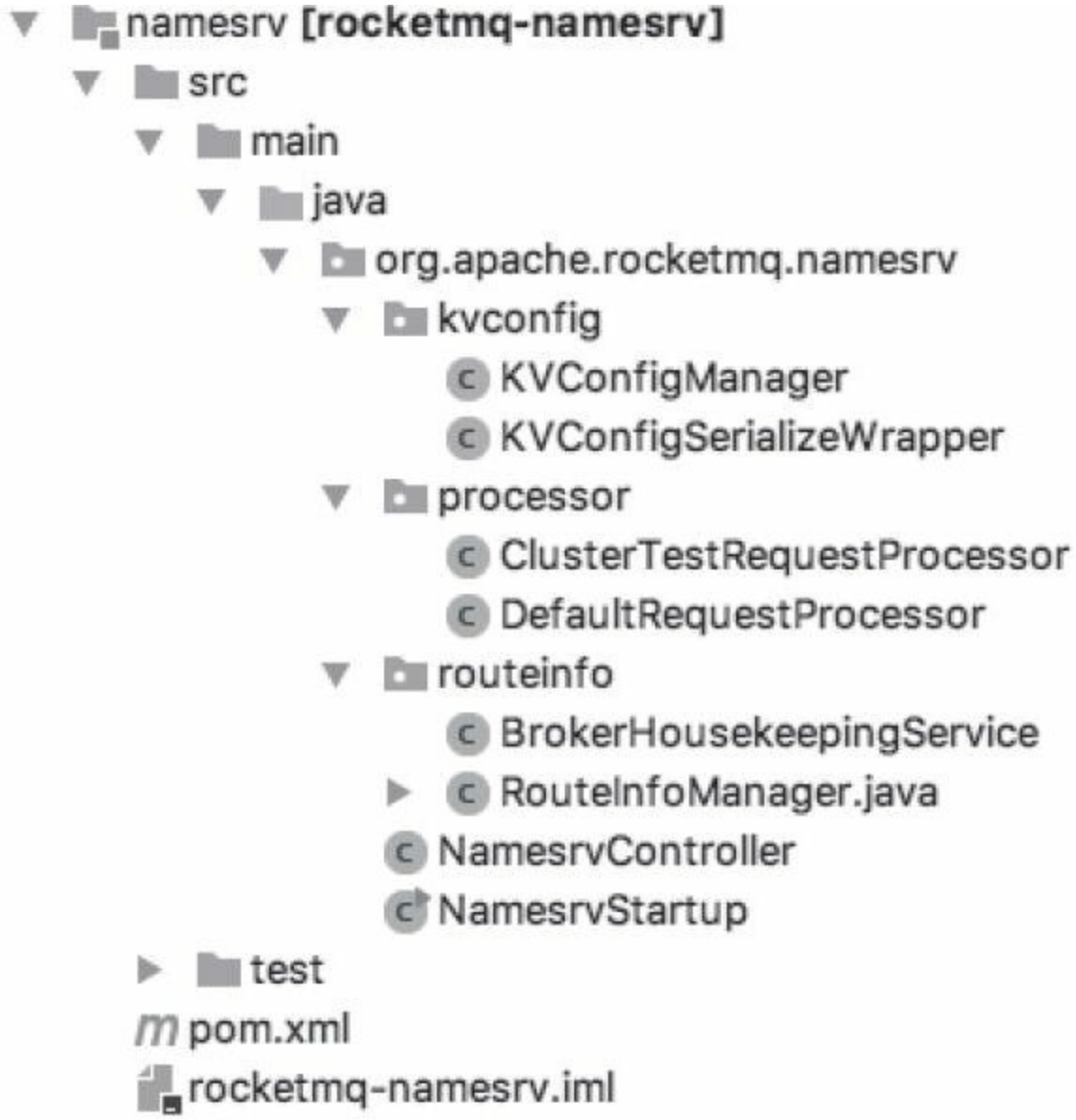


图10-1　NameServer源码目录

10.1.2　解析命令行参数

main0函数主要完成两个功能，第一个功能是解析命令行参数，我们通过源码来看一看，重点是解析-c和-p参数，如代码清单10-1所示。

代码清单10-1　解析NameServer命令行参数

```
Options options = ServerUtil.buildCommandlineOptions(new Options());
commandLine = ServerUtil.parseCmdLine("mqnamesrv", args,
    buildCommandlineOptions(options), new PosixParser());
if (null == commandLine) {
    System.exit(-1);
    return null;
}
final NamesrvConfig namesrvConfig = new NamesrvConfig();
final NettyServerConfig nettyServerConfig = new NettyServerConfig();
nettyServerConfig.setListenPort(9876);
if (commandLine.hasOption('c')) {
    String file = commandLine.getOptionValue('c');
    if (file != null) {
        InputStream in = new BufferedInputStream(new
            FileInputStream(file));
        properties = new Properties();
        properties.load(in);
        MixAll.properties2Object(properties, namesrvConfig);
        MixAll.properties2Object(properties, nettyServerConfig);
        namesrvConfig.setConfigStorePath(file);
        System.out.printf("load config properties file OK, " +
            file + "%n");
        in.close();
    }
}
if (commandLine.hasOption('p')) {
    MixAll.printObjectProperties(null, namesrvConfig);
    MixAll.printObjectProperties(null, nettyServerConfig);
    System.exit(0);
}
```

-c命令行参数用来指定配置文件的位置；-p命令行参数用来打印所有配置项的值。注意，用-p参数打印配置项的值之后程序就退出了，这是一个帮助调试的选项。

10.1.3　初始化NameServer的Controller

main0函数的另外一个功能是初始化Controller，如代码清单10-2所示。

代码清单10-2　初始化并启动Controller

```
// remember all configs to prevent discard
controller.getConfiguration().registerConfig(properties);
boolean initResult = controller.initialize();
if (!initResult) {
    controller.shutdown();
    System.exit(-3);
}
Runtime.getRuntime().addShutdownHook(new ShutdownHookThread(log,
    new Callable<Void>() {
    @Override
    public Void call() throws Exception {
        controller.shutdown();
        return null;
    }
}));
controller.start();
```

根据解析出的配置参数，调用controller.initialize（）来初始化，然后调用controller.start（）让NameServer开始服务。

还有一个逻辑是注册ShutdownHookThread，当程序退出的时候会调用controller.shutdown来做退出前的清理工作。

10.2 NameServer的总控逻辑

NameServer的总控逻辑在NamesrvController.java代码中。NameServer是集群的协调者，它只是简单地接收其他角色报上来的状态，然后根据请求返回相应的状态。首先，NameserverController把执行线程池初始化好，如代码清单10-3所示。

代码清单10-3 线程池初始化

```
this.remotingExecutor =
    Executors.newFixedThreadPool(nettyServerConfig
        .getServerWorkerThreads(), new ThreadFactoryImpl
        ("RemotingExecutorThread_"));
this.registerProcessor();

this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
    @Override
    public void run() {
       NamesrvController.this.routeInfoManager.scanNotActiveBroker();
    }
}, 5, 10, TimeUnit.SECONDS);
this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
    @Override
    public void run() {
       NamesrvController.this.kvConfigManager.printAllPeriodically();
    }
}, 1, 10, TimeUnit.MINUTES);
```

启动了一个默认是8个线程的线程池（private int serverWorkerThreads=8），还有两个定时执行的线程，一个用来扫描失效的Broker（scanNotActiveBroker），另一个用来打印配置信息（printAllPeriodically）。

然后启动负责通信的服务remotingServer，remotingServer监听一些端口，收到Broker、Client等发过来的请求后，根据请求的命令，调用不同的Processor来处理。这些不同的处理逻辑被放到上面初始化的线程池中执行，如代码清单10-4所示。

代码清单10-4 启动通信服务，关联初始化的线程池

```
this.remotingServer = new NettyRemotingServer(this.nettyServerConfig,
    this.brokerHousekeepingService);
……
```

```
if (namesrvConfig.isClusterTest()) {
    this.remotingServer.registerDefaultProcessor(new
            ClusterTestRequestProcessor(this, namesrvConfig
            .getProductEnvName()),
        this.remotingExecutor);
} else {
    this.remotingServer.registerDefaultProcessor(new
        DefaultRequestProcessor(this), this.remotingExecutor);
}
```

---

remotingServer是基于Netty封装的一个网络通信服务，要了解remoting-Server需要先对Netty有个基本的认知，后面会单独介绍。

10.3 核心业务逻辑处理

NameServer的核心业务逻辑，在DefaultRequestProcessor.java中可以一目了然地看出。网络通信服务模块收到请求后，就调用这个Processor来处理，如代码清单10-5所示。

代码清单10-5 根据请求码调用相应的处理逻辑

```
switch (request.getCode()) {
    case RequestCode.PUT_KV_CONFIG:
        return this.putKVConfig(ctx, request);
    case RequestCode.GET_KV_CONFIG:
        return this.getKVConfig(ctx, request);
    case RequestCode.DELETE_KV_CONFIG:
        return this.deleteKVConfig(ctx, request);
    case RequestCode.REGISTER_BROKER:
        Version brokerVersion = MQVersion.value2Version(request
            .getVersion());
        if (brokerVersion.ordinal() >= MQVersion.Version
            .V3_0_11.ordinal()) {
            return this.registerBrokerWithFilterServer(ctx, request);
        } else {
            return this.registerBroker(ctx, request);
        }
    case RequestCode.UNREGISTER_BROKER:
        return this.unregisterBroker(ctx, request);
    case RequestCode.GET_ROUTEINTO_BY_TOPIC:
        return this.getRouteInfoByTopic(ctx, request);
    case RequestCode.GET_BROKER_CLUSTER_INFO:
        return this.getBrokerClusterInfo(ctx, request);
    case RequestCode.WIPE_WRITE_PERM_OF_BROKER:
        return this.wipeWritePermOfBroker(ctx, request);
    case RequestCode.GET_ALL_TOPIC_LIST_FROM_NAMESERVER:
        return getAllTopicListFromNameserver(ctx, request);
    case RequestCode.DELETE_TOPIC_IN_NAMESRV:
        return deleteTopicInNamesrv(ctx, request);
    case RequestCode.GET_KVLIST_BY_NAMESPACE:
        return this.getKVListByNamespace(ctx, request);
    case RequestCode.GET_TOPICS_BY_CLUSTER:
        return this.getTopicsByCluster(ctx, request);
    case RequestCode.GET_SYSTEM_TOPIC_LIST_FROM_NS:
        return this.getSystemTopicListFromNs(ctx, request);
    case RequestCode.GET_UNIT_TOPIC_LIST:
        return this.getUnitTopicList(ctx, request);
    case RequestCode.GET_HAS_UNIT_SUB_TOPIC_LIST:
        return this.getHasUnitSubTopicList(ctx, request);
    case RequestCode.GET_HAS_UNIT_SUB_UNUNIT_TOPIC_LIST:
        return this.getHasUnitSubUnUnitTopicList(ctx, request);
    case RequestCode.UPDATE_NAMESRV_CONFIG:
        return this.updateConfig(ctx, request);
    case RequestCode.GET_NAMESRV_CONFIG:
        return this.getConfig(ctx, request);
    default:
        break;
}
```

逻辑主体是个switch语句，根据RequestCode调用不同的函数来处理，从RequestCode可以了解到NameServer的主要功能，比如：
REGISTER_BROKER是在集群中新加入一个Broker机器；
GET_ROUTEINTO_BY_TOPIC是请求获取一个Topic的路由信息；
WIPE_WRITE_PERM_OF_BROKER是删除一个Broker的写权限。

10.4 集群状态存储

NameServer作为集群的协调者，需要保存和维护集群的各种元数据，这是通过RouteInfoManager类来实现的，如代码清单10-6所示。

代码清单10-6 RouteInfoManager的存储结构

```
private final HashMap<String/* topic */, List<QueueData>> topicQueue-Table;
private final HashMap<String/* brokerName */, BrokerData> brokerAddr-Table;
private final HashMap<String/* clusterName */, Set<String/* brokerName
*/>> clusterAddrTable;
private final HashMap<String/* brokerAddr */, BrokerLiveInfo>
    brokerLiveTable;
private final HashMap<String/* brokerAddr */, List<String>/* Filter
Server */> filterServerTable;
public RouteInfoManager() {
    this.topicQueueTable = new HashMap<String, List<QueueData>>(1024);
    this.brokerAddrTable = new HashMap<String, BrokerData>(128);
    this.clusterAddrTable = new HashMap<String, Set<String>>(32);
    this.brokerLiveTable = new HashMap<String, BrokerLiveInfo>(256);
    this.filterServerTable = new HashMap<String, List<String>>(256);
}
```

每个结构存储着一类集群信息，具体含义在第5章有介绍。了解RocketMQ各个角色的功能后，对每个结构的处理逻辑就好理解了。下面重点看一下控制访问这些结构的锁机制。

锁分为互斥锁、读写锁；也可分为可重入锁、不可重入锁。在NameServer的场景中，读取操作多，更改操作少，所以选择读写锁能大大提高效率。对于如何选择可重入和不可重入锁，重点看函数间的调用关系，比如多次获取锁的示例代码，如果这个lock是不可重入的，代码无法正常执行，如代码清单10-7所示。

代码清单10-7 多次获取锁示例

```
Lock lock = new Lock();
public void outer() {
    lock.lock();
    inner();
    lock.unlock();
}
public void inner() {
    lock.lock();
    //do something lock.unlock(); }
}
```

RouteInfoManager中使用的是可重入的读写锁（private final ReadWriteLock lock=new ReentrantReadWriteLock（））， 我们以deleteTopic函数为例，看一下锁的使用方式，如代码清单10-8所示。

代码清单10-8　锁的使用方式

```
public void deleteTopic(final String topic) {
    try {
        try {
            this.lock.writeLock().lockInterruptibly();
            this.topicQueueTable.remove(topic);
        } finally {
            this.lock.writeLock().unlock();
        }
    } catch (Exception e) {
        log.error("deleteTopic Exception", e);
    }
}
```

首先锁的获取和执行逻辑要放到一个try{}里，然后在finally{}中释放。这是一种典型的使用方式，我们可以参考这种方式实现自己的代码。

10.5 本章小结

本章分析了NameServer模块的源码，NameServer是一个功能重要但是代码量不大的模块，所以选择这个模块入手，比较容易理解。我们在分析源码时，认真读懂一个模块后就可以对作者的代码风格、设计偏好等有基本的了解。下一章将分析Client模块的源码，我们使用RocketMQ时经常需要和Client模块打交道。

第11章　最常用的消费类

编写程序消费RocketMQ中消息的时候，最常用的类是DefaultMQPush-Consumer，这个类让我们消费消息变得很简单，这个类到底默默地为我们做了哪些事情呢？本章将对其做详细分析。

11.1 整体流程

我们使用DefaultMQPushConsumer的时候，一般流程是设置好GroupName、NameServer地址，以及订阅的Topic名称，然后填充Message处理函数，最后调用start（）。本节基于这个流程来分析源码。

11.1.1 上层接口类

DefaultMQPushConsumer类在org.apache.rocketmq.client.consumer包中，这个类担任着上层接口的角色，具体实现都在DefaultMQPushConsumerImpl类中，如代码清单11-1所示。

代码清单11-1 DefaultMQPushConsumer接口类

```
/**
 * Default constructor.
 */
public DefaultMQPushConsumer() {
    this(MixAll.DEFAULT_CONSUMER_GROUP, null, new
        AllocateMessageQueueAveragely());
}
/**
 * Constructor specifying consumer group, RPC hook and message queue
 * allocating algorithm.
 *
 * @param consumerGroup Consume queue.
 * @param rpcHook RPC hook to execute before each remoting command.
 * @param allocateMessageQueueStrategy message queue allocating algorithm.
 */
public DefaultMQPushConsumer(final String consumerGroup, RPCHook rpcHook,
    AllocateMessageQueueStrategy allocateMessageQueueStrategy) {
    this.consumerGroup = consumerGroup;
    this.allocateMessageQueueStrategy = allocateMessageQueueStrategy;
    defaultMQPushConsumerImpl = new DefaultMQPushConsumerImpl(this,
        rpcHook);
}
/**
 * Constructor specifying RPC hook.
 *
 * @param rpcHook RPC hook to execute before each remoting command.
 */
public DefaultMQPushConsumer(RPCHook rpcHook) {
    this(MixAll.DEFAULT_CONSUMER_GROUP, rpcHook, new
        AllocateMessageQueueAveragely());
}

/**
 * Constructor specifying consumer group.
 *
 * @param consumerGroup Consumer group.
 */
public DefaultMQPushConsumer(final String consumerGroup) {
    this(consumerGroup, null, new AllocateMessageQueueAveragely());
}
```

我们常用的是最后这个构造函数，只传入一个consumer Group名称作为参数，这个构造函数会把RPCHoop设为空，把负载均衡策略设置成平均策略。在构造函数的实现中，主要工作是创建

DefaultMQPushConsumerImpl对象。

11.1.2 DefaultMQPushConsumer的实现者

DefaultMQPushConsumerImpl具体实现了DefaultMQPushConsumer的业务逻辑，DefaultMQPushConsumerImpl.java在org.apache.rocketmq.client.impl.consumer这个包里，本节接下来从start方法着手分析。

首先是初始化MQClientInstance，并且设置好rebalance策略和pullApi-Wraper，有这些结构后才能发送pull请求获取消息，如代码清单11-2所示。

代码清单11-2 初始化MQClientInstance和pullApiWraper

```
this.mQClientFactory = MQClientManager.getInstance()
    .getAndCreateMQClientInstance(this.defaultMQPushConsumer,
        this.rpcHook);
this.rebalanceImpl.setConsumerGroup(this
    .defaultMQPushConsumer.getConsumerGroup());
this.rebalanceImpl.setMessageModel(this.defaultMQPushConsumer
    .getMessageModel());
this.rebalanceImpl.setAllocateMessageQueueStrategy(this
    .defaultMQPushConsumer.getAllocateMessageQueueStrategy());
this.rebalanceImpl.setmQClientFactory(this.mQClientFactory);
this.pullAPIWrapper = new PullAPIWrapper(
    mQClientFactory,
    this.defaultMQPushConsumer.getConsumerGroup(), isUnitMode
    ());
this.pullAPIWrapper.registerFilterMessageHook
    (filterMessageHookList);
```

然后是确定OffsetStore。OffsetStore里存储的是当前消费者所消费的消息在队列中的偏移量，如代码清单11-3所示。

代码清单11-3 初始化OffsetStore

```
if (this.defaultMQPushConsumer.getOffsetStore() != null) {
    this.offsetStore = this.defaultMQPushConsumer
        .getOffsetStore();
} else {
    switch (this.defaultMQPushConsumer.getMessageModel()) {
        case BROADCASTING:
            this.offsetStore = new LocalFileOffsetStore(this
                .mQClientFactory, this.defaultMQPushConsumer
                .getConsumerGroup());
            break;
        case CLUSTERING:
```

```
                this.offsetStore = new RemoteBrokerOffsetStore
                    (this.mQClientFactory, this
                        .defaultMQPushConsumer.getConsumerGroup());
                break;
            default:
                break;
        }
        this.defaultMQPushConsumer.setOffsetStore(this.offsetStore);
    }
    this.offsetStore.load();
```

根据消费消息方式的不同，OffsetStore的类型也不同。如果是BROADCASTING模式，使用的是LocalFileOffsetStore，Offset存到本地；如果是CLUSTERING模式，使用的是RemoteBrokerOffsetStore，Offset存到Broker机器上。

然后是初始化consumeMessageService，根据对消息顺序需求的不同，使用不同的Service类型，如代码清单11-4所示。

代码清单11-4　初始化consumeMessageService

```
if (this.getMessageListenerInner() instanceof
    MessageListenerOrderly) {
    this.consumeOrderly = true;
    this.consumeMessageService =
        new ConsumeMessageOrderlyService(this,
            (MessageListenerOrderly) this
                .getMessageListenerInner());
} else if (this.getMessageListenerInner() instanceof
    MessageListenerConcurrently) {
    this.consumeOrderly = false;
    this.consumeMessageService =
        new ConsumeMessageConcurrentlyService(this,
            (MessageListenerConcurrently) this
                .getMessageListenerInner());
}
this.consumeMessageService.start();
```

最后调用MQClientInstance的start方法，开始获取数据。

11.1.3 获取消息逻辑

获取消息的逻辑实现在public void pullMessage（final PullRequest pullRequest）函数中，这是一个很大的函数，前半部分是进行一些判断，是进行流量控制的逻辑（见代码清单11-5）；中间是对返回消息结果做处理的逻辑；后面是发送获取消息请求的逻辑。

代码清单11-5 流量控制逻辑

```
if (cachedMessageCount > this.defaultMQPushConsumer
    .getPullThresholdForQueue()) {
    this.executePullRequestLater(pullRequest,
        PULL_TIME_DELAY_MILLS_WHEN_FLOW_CONTROL);
    if ((queueFlowControlTimes++ % 1000) == 0) {
        log.warn(
            "the cached message count exceeds the threshold {}, so do" +
                " flow control, minOffset={}, maxOffset={}, count={}," +
                " size={} MiB, pullRequest={}, flowControlTimes={}",
            this.defaultMQPushConsumer.getPullThresholdForQueue(),
            processQueue.getMsgTreeMap().firstKey(), processQueue
                .getMsgTreeMap().lastKey(), cachedMessageCount,
            cachedMessageSizeInMiB, pullRequest, queueFlowControlTimes);
    }
    return;
}
if (cachedMessageSizeInMiB > this.defaultMQPushConsumer
    .getPullThresholdSizeForQueue()) {
    this.executePullRequestLater(pullRequest,
        PULL_TIME_DELAY_MILLS_WHEN_FLOW_CONTROL);
    if ((queueFlowControlTimes++ % 1000) == 0) {
        log.warn(
            "the cached message size exceeds the threshold {} MiB, so" +
                " do flow control, minOffset={}, maxOffset={}, " +
                "count={}, size={} MiB, pullRequest={}, " +
                "flowControlTimes={}",
            this.defaultMQPushConsumer.getPullThresholdSizeForQueue()
            , processQueue.getMsgTreeMap().firstKey(), processQueue
                .getMsgTreeMap().lastKey(), cachedMessageCount,
            cachedMessageSizeInMiB, pullRequest, queueFlowControlTimes);
    }
    return;
}
```

通过判断未处理消息的个数和总大小来控制是否继续请求消息。对于顺序消息还有一些特殊判断逻辑。获取的消息返回后，根据返回状态，调用相应的处理方法，如代码清单11-6所示。

代码清单11-6 对返回消息结果做处理

```
switch (pullResult.getPullStatus()) {
    case FOUND:
        long prevRequestOffset = pullRequest
            .getNextOffset();
        pullRequest.setNextOffset(pullResult
            .getNextBeginOffset());
        ……
        break;
    case NO_NEW_MSG:
        pullRequest.setNextOffset(pullResult
            .getNextBeginOffset());
        DefaultMQPushConsumerImpl.this.correctTagsOffset
            (pullRequest);
        DefaultMQPushConsumerImpl.this
            .executePullRequestImmediately(pullRequest);
        break;
    case NO_MATCHED_MSG:
        pullRequest.setNextOffset(pullResult
            .getNextBeginOffset());
        DefaultMQPushConsumerImpl.this.correctTagsOffset
            (pullRequest);
        DefaultMQPushConsumerImpl.this
            .executePullRequestImmediately(pullRequest);
        break;
    case OFFSET_ILLEGAL:
        log.warn("the pull request offset illegal, {} {}",
            pullRequest.toString(), pullResult.toString());
        pullRequest.setNextOffset(pullResult
            .getNextBeginOffset());
……        break;
    default:
        break;
}
```

最后是发送获取消息请求，这三个阶段不停地循环执行，直到程序被停止，如代码清单11-7所示。

代码清单11-7　发送pull请求

```
try {
    this.pullAPIWrapper.pullKernelImpl(
        pullRequest.getMessageQueue(),
        subExpression,
        subscriptionData.getExpressionType(),
        subscriptionData.getSubVersion(),
        pullRequest.getNextOffset(),
        this.defaultMQPushConsumer.getPullBatchSize(),
        sysFlag,
        commitOffsetValue,
        BROKER_SUSPEND_MAX_TIME_MILLIS,
        CONSUMER_TIMEOUT_MILLIS_WHEN_SUSPEND,
        CommunicationMode.ASYNC,
        pullCallback
    );
} catch (Exception e) {
    log.error("pullKernelImpl exception", e);
    this.executePullRequestLater(pullRequest,
        PULL_TIME_DELAY_MILLS_WHEN_EXCEPTION);
}
```

11.2 消息的并发处理

本节重点看一下实现消息并发处理的代码，并发处理会增大实现流量控制、保证消息顺序方面的难度。

11.2.1 并发处理过程

处理效率的高低是反应Consumer实现好坏的重要指标，本节以Consume-MessageConcurrentlyService类为例来分析RocketMQ的实现方式。Consume-MessageConcurrentlyService类在org.apache.rocketmq.client.impl.consumer包中。

这个类定义了三个线程池，一个主线程池用来正常执行收到的消息，用户可以自定义通过consumeThreadMin和consumeThreadMax来自定义线程个数。另外两个都是单线程的线程池，一个用来执行推迟消费的消息，另一个用来定期清理超时消息（15分钟），如代码清单11-8所示。

代码清单11-8　三个线程池

```
this.consumeExecutor = new ThreadPoolExecutor(
    this.defaultMQPushConsumer.getConsumeThreadMin(),
    this.defaultMQPushConsumer.getConsumeThreadMax(), 1000 * 60,
    TimeUnit.MILLISECONDS, this.consumeRequestQueue,
    new ThreadFactoryImpl("ConsumeMessageThread_"));
this.scheduledExecutorService =
    Executors.newSingleThreadScheduledExecutor(new ThreadFactoryImpl(
        "ConsumeMessageScheduledThread_"));
this.cleanExpireMsgExecutors =
    Executors.newSingleThreadScheduledExecutor(new ThreadFactoryImpl(
        "CleanExpireMsgScheduledThread_"));
```

从Broker获取到一批消息以后，根据BatchSize的设置，把一批消息封装到一个ConsumeRequest中，然后把这个ConsumeRequest提交到consumeExecutor线程池中执行，如代码清单11-9所示。

代码清单11-9　任务分发逻辑

```
if (msgs.size() <= consumeBatchSize) {
    ConsumeRequest consumeRequest = new ConsumeRequest(msgs,
        processQueue, messageQueue);
    try {
        this.consumeExecutor.submit(consumeRequest);
    } catch (RejectedExecutionException e) {
        this.submitConsumeRequestLater(consumeRequest);
    }
} else {
    for (int total = 0; total < msgs.size(); ) {
```

```
            List<MessageExt> msgThis = new ArrayList<MessageExt>
                (consumeBatchSize);
            for (int i = 0; i < consumeBatchSize; i++, total++) {
                if (total < msgs.size()) {
                    msgThis.add(msgs.get(total));
                } else {
                    break;
                }
            }
            ConsumeRequest consumeRequest = new ConsumeRequest(msgThis,
                processQueue, messageQueue);
            try {
                this.consumeExecutor.submit(consumeRequest);
            } catch (RejectedExecutionException e) {
                for (; total < msgs.size(); total++) {
                    msgThis.add(msgs.get(total));
                }

                this.submitConsumeRequestLater(consumeRequest);
            }
        }
    }
```

消息的处理结果可能有不同的值，主要的两个是CONSUME_SUCCESS和RECONSUME_LATER。如果消费不成功，要把消息提交到上面说的scheduledExecutorService线程池中，5秒后再执行；如果消费模式是CLUSTERING模式，未消费成功的消息会先被发送回Broker，供这个ConsumerGroup里的其他Consumer消费，如果发送回Broker失败，再调用RECONSUME_LATER，消息消费的Status处理逻辑如代码清单11-10所示。

代码清单11-10　消息消费的Status处理逻辑

```
    switch (this.defaultMQPushConsumer.getMessageModel()) {
        case BROADCASTING:
            for (int i = ackIndex + 1; i < consumeRequest.getMsgs().size
                (); i++) {
                MessageExt msg = consumeRequest.getMsgs().get(i);
                log.warn("BROADCASTING, the message consume failed, drop " +
                    "it, {}", msg.toString());
            }
            break;
        case CLUSTERING:
            List<MessageExt> msgBackFailed = new ArrayList<MessageExt>
                (consumeRequest.getMsgs().size());
            for (int i = ackIndex + 1; i < consumeRequest.getMsgs().size
                (); i++) {
                MessageExt msg = consumeRequest.getMsgs().get(i);
                boolean result = this.sendMessageBack(msg, context);
                if (!result) {
                    msg.setReconsumeTimes(msg.getReconsumeTimes() + 1);
                    msgBackFailed.add(msg);
                }
            }
            if (!msgBackFailed.isEmpty()) {
                consumeRequest.getMsgs().removeAll(msgBackFailed);
```

```
                this.submitConsumeRequestLater(msgBackFailed,
                    consumeRequest.getProcessQueue(), consumeRequest
                        .getMessageQueue());
            }
            break;
        default:
            break;
    }
```

处理逻辑是用户自定义的，当消息量大的时候，处理逻辑执行效率的高低影响系统的吞吐量。可以把多条消息组合起来处理，或者提高线程数，以提高系统的吞吐量。

11.2.2 ProcessQueue对象

在前面的源码中，有个ProcessQueue类型的对象，这个对象的功能是什么呢？从Broker获得的消息，因为是提交到线程池里并行执行，很难监控和控制执行状态，比如如何获得当前消息堆积的数量，如何解决处理超时情况等。RocketMQ定义了一个快照类ProcessQueue来解决这些问题，在PushConsumer运行的时候，每个Message Queue都会有一个对应的ProcessQueue对象，保存了这个Message Queue消息处理状态的快照，如代码清单11-11所示。

ProcessQueue对象里主要的内容是一个TreeMap和一个读写锁。TreeMap里以Message Queue的Offset作为Key，以消息内容的引用为Value，保存了所有从MessageQueue获取到但是还未被处理的消息，读写锁控制着多个线程对TreeMap对象的并发访问。

代码清单11-11　保存消息消费的状态

```
private final ReadWriteLock lockTreeMap = new ReentrantReadWriteLock();
private final TreeMap<Long, MessageExt> msgTreeMap = new TreeMap<Long, MessageExt>(
private final AtomicLong msgCount = new AtomicLong();
private final AtomicLong msgSize = new AtomicLong();
private final Lock lockConsume = new ReentrantLock();
```

有了ProcessQueue对象，可以随时停止、启动消息的消费，同时也可用于帮助实现顺序消费消息。顺序消息是通过ConsumeMessageOrderlyService类实现的，主要流程和ConsumeMessageConcurrentlyService类似，区别只是在对并发消费的控制上。

11.3 生产者消费者的底层类

无论是生产者还是消费者，在底层都要和Broker打交道，进行消息收发。在源码层面，底层的功能被抽象成同一个类，负责和Broker打交道，本节详细介绍这个类的情况。

11.3.1 MQClientInstance类的创建规则

MQClientInstance是客户端各种类型的Consumer和Producer的底层类。这个类首先从NameServer获取并保存各种配置信息，比如Topic的Route信息。同时MQClientInstance还会通过MQClientAPIImpl类实现消息的收发，也就是从Broker获取消息或者发送消息到Broker。

既然MQClientInstance实现的是底层通信功能和获取并保存元数据的功能，就没必要每个Consumer或Producer都创建一个对象，一个MQClientInstance对象可以被多个Consumer或Producer公用。RocketMQ通过一个工厂类达到共用MQClientInstance的目的。MQClientInstance的创建如代码清单11-12所示。

代码清单11-12　创建MQClientInstance

```
MQClientManager.getInstance().getAndCreateMQClientInstance(this.defaultMQProducer, r
```

注意，MQClientInstance是通过工厂类被创建的，并不是一个单例模式，有些情况下需要创建多个实例。首先来看看MQClientInstance的创建规则，如代码清单11-13所示。

代码清单11-13　MQClientInstance创建规则

```
public MQClientInstance getAndCreateMQClientInstance(
    final ClientConfig clientConfig, RPCHook rpcHook) {
    String clientId = clientConfig.buildMQClientId();
    MQClientInstance instance = this.factoryTable.get(clientId);
    if (null == instance) {
        instance =
            new MQClientInstance(clientConfig.cloneClientConfig(),
                this.factoryIndexGenerator.getAndIncrement(), clientId,
                rpcHook);
        MQClientInstance prev = this.factoryTable.putIfAbsent(clientId,
            instance);
        if (prev != null) {
            instance = prev;
            log.warn("Returned Previous MQClientInstance for " +
                "clientId:[{}]", clientId);
        } else {
            log.info("Created new MQClientInstance for clientId:[{}]",
                clientId);
        }
    }
```

```
        return instance;
    }
```

系统中维护了ConcurrentMap<String/*clientId*/，MQClientInstance>factoryTable这个Map对象，每创建一个新的MQClientInstance，都会以clientId作为Key放入Map结构中。clientId的格式是“clientIp”+@+“InstanceName”，其中clientIp是客户端机器的IP地址，一般不会变，instancename有默认值，也可以被手动设置。

普通情况下，一个用到RocketMQ客户端的Java程序，或者说一个JVM进程只要有一个MQClientInstance实例就够了。这时候创建一个或多个Consumer或者Producer，底层使用的是同一个MQClientInstance实例。

在quick start文档中创建一个DefaultMQPushConsumer来接收消息，没有设置这个Consumer的InstanceName参数（通过setInstanceName函数进行设置），这个时候InstanceName的值是默认的“DEFAULT”。实际创建的MQClientInstance个数由设定的逻辑进行控制。InstanceName的生成逻辑如代码清单11-14所示。

代码清单11-14　InstanceName生成逻辑

```
    if (this.defaultMQPushConsumer.getMessageModel() == MessageModel.CLUSTERING) {
        this.defaultMQPushConsumer.changeInstanceNameToPID();
    }
    public void changeInstanceNameToPID() {
        if (this.instanceName.equals("DEFAULT")) {
            this.instanceName = String.valueOf(UtilAll.getPid());
        }
    }
```

从InstanceName的创建逻辑就可以看出，如果创建Consumer或者Producer类型的时候不手动指定InstanceName，进程中只会有一个MQClientInstance对象。

有些情况下只有一个MQClientInstance对象是不够的，比如一个Java程序需要连接两个RoceketMQ集群，从一个集群读取消息，发送到另一个集群，一个MQClientInstance对象无法支持这种场景。这种情况下一定要手动指定不同的InstanceName，底层会创建两个MQClientInstance对象。

11.3.2 MQClientInstance类的功能

首先来看一下MQClientInstance类的Start函数，从Start函数中的逻辑能大致了解MQClientInstance类的功能，如代码清单11-15所示。

代码清单11-15 MQClientInstance类Start函数

```
public void start() throws MQClientException {
    synchronized (this) {
        switch (this.serviceState) {
            case CREATE_JUST:
                this.serviceState = ServiceState.START_FAILED;
                // If not specified,looking address from name server
                if (null == this.clientConfig.getNamesrvAddr()) {
                    this.mQClientAPIImpl.fetchNameServerAddr();
                }
                // Start request-response channel
                this.mQClientAPIImpl.start();
                // Start various schedule tasks
                this.startScheduledTask();
                // Start pull service
                this.pullMessageService.start();
                // Start rebalance service
                this.rebalanceService.start();
                // Start push service
                this.defaultMQProducer.getDefaultMQProducerImpl().start (false);
                log.info("the client factory [{}] start OK", this.clientId);
                this.serviceState = ServiceState.RUNNING;
                break;
            case RUNNING:
                break;
            case SHUTDOWN_ALREADY:
                break;
            case START_FAILED:
                throw new MQClientException("The Factory object[" + this.getClientId
            default:
                break;
        }
    }
}
```

Start函数中的MQClientAPIImpl对象用来负责底层消息通信，然后启动pullMessageService和rebalanceService。在类的成员变量中，用topicRouteTable、brokerAddrTable等来存储从NameServer中获得的集群状态信息，并通过一个ScheduledTask来维护这些信息。MQClientInstance中定时执行的任务如代码清单11-16所示。

代码清单11-16 MQClientInstance中定时执行的任务

```
private void startScheduledTask() {
    if (null == this.clientConfig.getNamesrvAddr()) {
        this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                try {
                    MQClientInstance.this.mQClientAPIImpl
                        .fetchNameServerAddr();
                } catch (Exception e) {
                    log.error("ScheduledTask fetchNameServerAddr " +
                        "exception", e);
                }
            }
        }, 1000 * 10, 1000 * 60 * 2, TimeUnit.MILLISECONDS);
    }
    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            try {
                MQClientInstance.this.updateTopicRouteInfoFromNameServer();
            } catch (Exception e) {
                log.error("ScheduledTask " +
                    "updateTopicRouteInfoFromNameServer exception", e);
            }
        }
    }, 10, this.clientConfig.getPollNameServerInterval(), TimeUnit
        .MILLISECONDS);
    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            try {
                MQClientInstance.this.cleanOfflineBroker();
                MQClientInstance.this.sendHeartbeatToAllBrokerWithLock();
            } catch (Exception e) {
                log.error("ScheduledTask sendHeartbeatToAllBroker " +
                    "exception", e);
            }
        }
    }, 1000, this.clientConfig.getHeartbeatBrokerInterval(), TimeUnit
        .MILLISECONDS);

    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            try {
                MQClientInstance.this.persistAllConsumerOffset();
            } catch (Exception e) {
                log.error("ScheduledTask persistAllConsumerOffset " +
                    "exception", e);
            }
        }
    }, 1000 * 10, this.clientConfig.getPersistConsumerOffsetInterval(),
        TimeUnit.MILLISECONDS);
    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            try {
                MQClientInstance.this.adjustThreadPool();
            } catch (Exception e) {
                log.error("ScheduledTask adjustThreadPool exception", e);
            }
        }
    }, 1, 1, TimeUnit.MINUTES);
}
```

从代码中可以看出，MQClientInstance会定时进行如下几个操作：获取NameServer地址、更新TopicRoute信息、清理离线的Broker和保存消费者的Offset。

11.4 本章小结

本章分析的是Client模块里的代码，我们在使用RocketMQ的时候，更多的是和这个模块里的代码打交道。本章重点分析了DefaultMQPushConsumerImpl类，然后分析了Consumer的并发处理过程，最后分析了客户端Class统一的底层通信类MQClientInstance。下一章将从代码层面分析RocketMQ的主从同步机制。

第12章　主从同步机制

RocketMQ的Broker分为Master和Slave两个角色，为了保证高可用性，Master角色的机器接收到消息后，要把内容同步到Slave机器上，这样一旦Master宕机，Slave机器依然可以提供服务。本章分析Master和Slave角色机器间同步功能实现的源码。

12.1 同步属性信息

Slave需要和Master同步的不只是消息本身，一些元数据信息也需要同步，比如TopicConfig信息、ConsumerOffset信息、DelayOffset和SubscriptionGroupConfig信息。Broker在启动的时候，判断自己的角色是否是Slave，是的话就启动定时同步任务，如代码清单12-1所示。

代码清单12-1 Slave角色定时同步元数据信息

```
if (BrokerRole.SLAVE == this.messageStoreConfig.getBrokerRole()) {
    if (this.messageStoreConfig.getHaMasterAddress() != null && this.messageStoreCor
        this.messageStore.updateHaMasterAddress(this.messageStoreConfig.getHaMasterA
        this.updateMasterHAServerAddrPeriodically = false;
    } else {
        this.updateMasterHAServerAddrPeriodically = true;
    }
    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            try {
                BrokerController.this.slaveSynchronize.syncAll();
            } catch (Throwable e) {
                log.error("ScheduledTask syncAll slave exception", e);
            }
        }
    }, 1000 * 10, 1000 * 60, TimeUnit.MILLISECONDS);
}
```

在syncAll函数里，调用syncTopicConfig（）、syncConsumerOffset（）、syncDelayOffset（）和syncSubscriptionGroupConfig（）进行元数据同步。我们以syncConsumerOffset为例，来看看底层的具体实现，如代码清单12-2所示。

代码清单12-2 syncConsumerOffset具体实现

```
public ConsumerOffsetSerializeWrapper getAllConsumerOffset(
    final String addr) throws InterruptedException, RemotingTimeoutException,
    RemotingSendRequestException, RemotingConnectException, MQBroker-Exception {
    RemotingCommand request = RemotingCommand.createRequestCommand(RequestCode.GET_A
    RemotingCommand response = this.remotingClient.invokeSync(addr, request, 3000);
    assert response != null;
    switch (response.getCode()) {
        case ResponseCode.SUCCESS: {
            return ConsumerOffsetSerializeWrapper.decode(response.getBody(), Consume
        }
```

```
            default:
                break;
        }
        throw new MQBrokerException(response.getCode(), response.getRemark());
    }
```

---

sysConsumer Offset（）的基本逻辑是组装一个RemotingCommand，底层通过Netty将消息发送到Master角色的Broker，然后获取Offset信息。

12.2 同步消息体

本节介绍Master和Slave之间同步消息体内容的方法，也就是同步CommitLog内容的方法。CommitLog和元数据信息不同：首先，CommitLog的数据量比元数据要大；其次，对实时性和可靠性要求也不一样。元数据信息是定时同步的，在两次同步的时间差里，如果出现异常可能会造成Master上的元数据内容和Slave上的元数据内容不一致，不过这种情况还可以补救（手动调整Offset，重启Consumer等）。CommitLog在高可靠性场景下如果没有及时同步，一旦Master机器出故障，消息就彻底丢失了。所以有专门的代码来实现Master和Slave之间消息体内容的同步。

主要的实现代码在Broker模块的org.apache.rocketmq.store.ha包中，里面包括HAService、HAConnection和WaitNotifyObject这三个类。

HAService是实现commitLog同步的主体，它在Master机器和Slave机器上执行的逻辑不同，默认是在Master机器上执行，见代码清单12-3。

代码清单12-3　根据Broker角色，确定是否设置HaMasterAddress

```
if (BrokerRole.SLAVE == this.messageStoreConfig.getBrokerRole()) {
    if (this.messageStoreConfig.getHaMasterAddress() != null && this.messageStoreCon
        .getHaMasterAddress().length() >= 6) {
        this.messageStore.updateHaMasterAddress(this.messageStoreConfig.getHaMasterA
        this.updateMasterHAServerAddrPeriodically = false;
    } else {
        this.updateMasterHAServerAddrPeriodically = true;
    }
```

当Broker角色是Slave的时候，MasterAddr的值会被正确设置，这样HAService在启动的时候，在HAClient这个内部类中，connectMaster会被正确执行，如代码清单12-4所示。

代码清单12-4　Slave角色连接Master

```
private boolean connectMaster() throws ClosedChannelException {
    if (null == socketChannel) {
        String addr = this.masterAddress.get();
        if (addr != null) {
```

```
                SocketAddress socketAddress = RemotingUtil.string2SocketAddress(addr);
                if (socketAddress != null) {
                    this.socketChannel = RemotingUtil.connect(socketAddress);
                    if (this.socketChannel != null) {
                        this.socketChannel.register(this.selector, SelectionKey.OP_READ)
                    }
                }
            }
            this.currentReportedOffset = HAService.this.defaultMessageStore.getMaxPhyOf

            this.lastWriteTimestamp = System.currentTimeMillis();
        }
        return this.socketChannel != null;
    }
```

从代码中可以看出，HAClient试图通过Java NIO函数去连接Master角色的Broker。Master角色有相应的监听代码，如代码清单12-5所示。

代码清单12-5　监听Slave的HA连接

```
    /**
     * Starts listening to slave connections.
     *
     * @throws Exception If fails.
     */
    public void beginAccept() throws Exception {
        this.serverSocketChannel = ServerSocketChannel.open();
        this.selector = RemotingUtil.openSelector();
        this.serverSocketChannel.socket().setReuseAddress(true);
        this.serverSocketChannel.socket().bind(this.socketAddressListen);
        this.serverSocketChannel.configureBlocking(false);
        this.serverSocketChannel.register(this.selector, SelectionKey.OP_ACCEPT);
    }
```

CommitLog的同步，不是经过netty command的方式，而是直接进行TCP连接，这样效率更高。连接成功以后，通过对比Master和Slave的Offset，不断进行同步。

12.3 sync_master和async_master

sync_master和async_master是写在Broker配置文件里的配置参数，这个参数影响的是主从同步的方式。从字面意思理解，sync_master是同步方式，也就是Master角色Broker中的消息要立刻同步过去；async_master是异步方式，也就是Master角色Broker中的消息是通过异步处理的方式同步到Slave角色的机器上的。下面结合代码来分析，sync_master下的消息同步如代码清单12-6所示。

代码清单12-6 sync_master下的消息同步

```
public void handleHA(AppendMessageResult result,
    PutMessageResult putMessageResult, MessageExt messageExt) {
    if (BrokerRole.SYNC_MASTER == this.defaultMessageStore
        .getMessageStoreConfig().getBrokerRole()) {
        HAService service = this.defaultMessageStore.getHaService();
        if (messageExt.isWaitStoreMsgOK()) {
            // Determine whether to wait
            if (service.isSlaveOK(result.getWroteOffset() + result
                .getWroteBytes())) {
                GroupCommitRequest request = new GroupCommitRequest
                    (result.getWroteOffset() + result
                    .getWroteBytes());
                service.putRequest(request);
                service.getWaitNotifyObject().wakeupAll();
                boolean flushOK =
                    request.waitForFlush(this.defaultMessageStore
                        .getMessageStoreConfig().getSyncFlushTimeout());
                if (!flushOK) {
                    log.error("do sync transfer other node, wait return, " +
                        "but failed, topic: " + messageExt
                        .getTopic() + " tags: "
                        + messageExt.getTags() + " client address: " +
                        messageExt.getBornHostNameString());
                    putMessageResult.setPutMessageStatus(PutMessageStatus
                        .FLUSH_SLAVE_TIMEOUT);
                }
            }
            // Slave problem
            else {
                // Tell the producer, slave not available
                putMessageResult.setPutMessageStatus(PutMessageStatus
                    .SLAVE_NOT_AVAILABLE);
            }
        }
    }
}
```

在CommitLog类的putMessage函数末尾，调用handleHA函数。代码中的关键词是wakeupAll和waitForFlush，在同步方式下，Master每次写

消息的时候，都会等待向Slave同步消息的过程，同步完成后再返回，如代码清单12-7所示。（putMessage函数比较长，仅列出关键的代码）。

代码清单12-7　putMessage中调用handleHA

```
public PutMessageResult putMessage(final MessageExtBrokerInner msg) {
    // Set the storage time
    msg.setStoreTimestamp(System.currentTimeMillis());
    // Set the message body BODY CRC (consider the most appropriate setting
    // on the client)
    msg.setBodyCRC(UtilAll.crc32(msg.getBody()));
    // Back to Results
    AppendMessageResult result = null;

    StoreStatsService storeStatsService = this.defaultMessageStore
        .getStoreStatsService();

    String topic = msg.getTopic();
    int queueId = msg.getQueueId();

  ……

    handleDiskFlush(result, putMessageResult, msg);
    handleHA(result, putMessageResult, msg);

    return putMessageResult;
}
```

12.4 本章小结

本章分析了Master和Slave角色的Broker之间同步信息功能的实现。需要同步的信息分为两种类型，实现方式各不相同：一种是元数据信息，采用基于Netty的command方式来同步消息；另一种是commitLog信息，同步方式是直接基于Java NIO来实现。下一章将介绍RocketMQ底层通信逻辑的具体实现。

第13章　基于Netty的通信实现

本章分析RocketMQ底层通信的实现机制，作为一个分布式消息队列，通信的质量至关重要。基于TCP协议和Socket实现一个高效、稳定的通信程序并不容易，有很多大大小小的“坑”等待着经验不足的开发者。RocketMQ选择不重复发明轮子，基于Netty库来实现底层的通信功能。

13.1 Netty介绍

Netty是一个网络应用框架，或者说是一个Java网络开发库。Netty提供异步事件驱动的方式，使用它可以快速地开发出高性能的网络应用程序，比如客户端/服务器自定义协议程序，大大简化了网络程序的开发过程。

Netty是一个精心设计的框架，它从许多协议实现中吸收了丰富的经验，比如FTP、SMTP、HTTP等许多基于二进制和文本的传统协议。借助Netty，可以比较容易地开发出达到Java网络专家+并发编程专家水平的通信程序。

了解Netty前需要对Java NIO有个基本的了解，熟悉Channel、ByteBuffer、Selector等基本概念。对于Java网络编程经验不多的读者，可以试着先用Java NIO的基本类写一个简单的Client/Server程序，然后再用Netty对比着实现一遍，这样比较容易理解Netty里各种组件存在的原因。

13.2 Netty架构总览

如图13-1所示，Netty主要分为三部分：一是底层的零拷贝技术和统一通信模型；二是基于JVM实现的传输层；三是常用协议支持。读者可以参考架构图做一个基本的了解，如果读者想深入了解的话可以阅读一些专门介绍Netty的书籍。

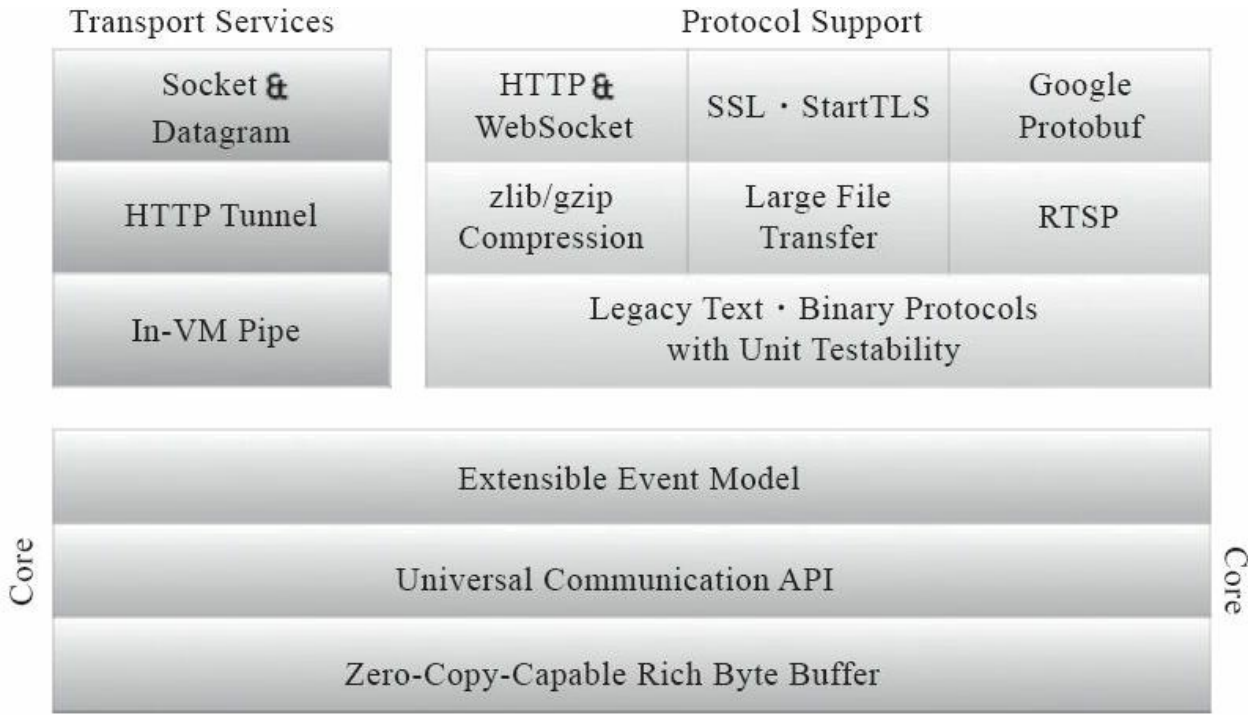


图13-1 Netty整体架构

13.2.1 重新实现ByteBuffer

在网络通信中，CPU处理数据的速度大大快于网络传输数据的速度，所以需要引入缓冲区，将网络传输的数据放入缓冲区，累积足够的数据再发给CPU处理。

Netty使用自己重新实现的buffer API，而不是使用NIO的ByteBuffer来表示一个连续的字节序列。新实现的buffer类型ByteBuf可以从底层解决ByteBuffer的一些问题，是一种更适合日常网络应用开发需要的缓存类型。重新实现的ByteBuf特性包括允许使用自定义的缓存类型、透明的零拷贝实现、比ByteBuffer更快的响应速度等。

字节缓存在网络通信中会被频繁地使用，ByteBuf实现的是一个非常轻量级的字节数组包装器。ByteBuf有读操作和写操作，为了便于用户使用，该缓冲区维护了读索引和写索引。ByteBuf由三个片段构成：废弃段、可读段和可写段。其中，可读段表示缓冲区实际存储的可用数据。当用户使用read或者skip方法时，将会增加读索引。读索引之前的数据将进入废弃段，表示该数据已被使用过了。此外，用户可主动使用discardReadBytes清空废弃段以便得到更多的可写空间。简单来说和ByteBuffer相比，ByteBuf用在网络编程时更合适，更易用。

13.2.2 统一的异步I/O接口

传统的Java I/O API在应对不同的传输协议时需要使用不同的类型和方法。例如java.net.Socket和java.net.DatagramSocket，但它们没有相同的父类型，因此需要使用不同的调用方式执行Socket操作。因为在模式上不匹配，所以更换网络应用的传输协议时工作会变得很繁杂。由于（Java I/O API）缺乏协议间的可移植性，无法在不修改网络传输层的前提下增加多种协议的支持。从理论上讲，多种应用层协议可运行在多种传输层协议之上，例如TCP/IP、UDP/IP、SCTP和串口通信。

还有个复杂的情况是，Java的新I/O（NIO）API与原有的阻塞式I/O（OIO）API不兼容。这两者无论是在设计上还是在性能上，其特性都不相同，可是在开发时一般只选择某一种API。例如，在用户数较小的时候可以选择使用传统的OIO（Old I/O）API，毕竟与NIO相比使用OIO更加容易；但是当业务快速增长，服务器需要同时处理成千上万的客户连接时问题就来了，这时候不得不尝试使用NIO来解决，新的NIO Selector编程接口和Old I/O差别很大，很难做到快速升级。

Netty有一个被称为Channel的统一异步I/O编程接口，这个编程接口抽象了所有点对点的通信操作。这样，如果应用是基于Netty的某一种传输方式来实现的，则可以快速迁移到另一种传输实现上。Netty提供了几种拥有相同编程接口的基本传输实现：

·基于NIO的TCP/IP传输（io.netty.channel.nio）；

·基于OIO的TCP/IP传输（io.netty.channel.oio）；

·基于OIO的UDP/IP传输（io.netty.channel.oio）；

·本地传输（io.netty.channel.local）。

切换不同的传输实现通常只需修改几行代码，而且由于核心API具有高度的可扩展性，很容易定制自己的传输实现。

13.2.3 基于拦截链模式的事件模型

一个定义良好并具有扩展能力的事件模型可以大大提高事件驱动程序的效率，Netty就具有定义良好的I/O事件模型，它采用严格的层次结构来区分不同的事件类型，Netty也允许在不破坏现有代码的情况下实现自己的事件类型。事件模型是Netty的一个亮点，很多NIO通信框架没有或者仅有有限的事件模型概念，当需要一个新的事件类型的时候常常需要修改已有的代码，有的甚至不允许进行自定义的扩展。

在Netty中，ChannelPipeline内部的一个ChannelEvent被一组ChannelHandler处理。这个管道是Intercepting Filter（拦截过滤器）模式的一种高级形式的实现，因此对于一个事件如何被处理，以及管道内部处理器间的交互过程，用户拥有绝对的控制力。

13.2.4 高级组件

Netty提供了一系列的高级组件来让开发过程更加快捷，比如Codec框架、SSL/TLS支持、HTTP实现等。

首先看看Codec框架。从业务逻辑代码中分离协议处理部分可以让代码结构变得更清晰，但是如果从零开始实现会有很高的复杂性，比如处理分段消息，相互叠加的多层协议，还有些协议复杂到无法在一台独立的状态机上实现。Netty提供了一组构建在其核心模块之上的codec实现，是一种可扩展、可重用、可单元测试，并且是多层的codec框架，为用户提供容易维护的codec代码。

Netty还提供对SSL/TLS的支持，不同于传统阻塞式的I/O实现，在NIO模式下支持SSL功能不能只是简单地包装一下流数据并进行加密或解密工作，还需要借助于javax.net.ssl.SSLEngine。SSLEngine是一个有状态的实现，使用SSLEngine必须管理所有可能的状态，例如密码套件、密钥协商（或重新协商）、证书交换以及认证等，而且SSLEngine不是一个绝对的线程安全实现。在Netty内部，SslHandler封装了所有艰难的细节，以及使用SSLEngine可能带来的陷阱。用户只需要配置并将该SslHandler插入你的ChannelPipeline中即可，而且Netty允许实现像StartTlS那样的高级特性。

HTTP是互联网上最受欢迎的协议，与现有的HTTP实现相比，Netty的HTTP实现是相当与众不同的。在HTTP消息的低层交互过程中用户拥有绝对的控制力，因为Netty的HTTP实现只是一些HTTP Codec和HTTP消息类的简单组合，不存在任何限制，例如那种被迫选择的线程模型。用户可以根据自己的需求编写那种可以完全按照你期望的工作方式工作的客户端或服务器端代码，比如线程模型、连接生命期、快编码等。基于这种高度可定制化的特性，用户可以开发一个非常高效的HTTP服务器，例如要求持久化链接以及服务器端推送技术的聊天服务，需要保持链接直至整个文件下载完成的媒体流服务，需要上传大文件并且没有内存压力的文件服务，支持大规模混合客户端应用用于连接以万计的第三方异步web服务等。

Netty的WebSockets实现，WebSockets允许双向，全双工通信信道。在TCP socket中，它被设计为允许一个Web浏览器和Web服务器之

间通过数据流交互。WebSocket协议已经被IETF列为RFC 6455规范，并且Netty实现了RFC 6455和一些老版本的规范。

此外Netty还支持Google Protocol Buffer，Google Protocol Buffers是快速实现一个高效的二进制协议的理想方案。通过使用ProtobufEncoder和ProtobufDecoder，我们可以把Google Protocol Buffers编译器（protoc）生成的消息类放入Netty的codec实现中。

13.3 Netty用法示例

13.3.1 Discard服务器

世上最简单的协议不是“Hello，World！”而是DISCARD服务器。这个协议会抛弃任何收到的数据而不响应。实现DISCARD协议只需忽略所有收到的数据。我们从Handler（处理器）的实现开始，Handler是由Netty生成用来处理I/O事件的，如代码清单13-1所示。

代码清单13-1 DiscardServerHandler实现

```
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
/**
 * 处理服务端 channel.
 */
public class DiscardServerHandler extends ChannelInboundHandlerAdapter { // (1)
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) { // (2)
        // 默默地丢弃收到的数据
        ((ByteBuf) msg).release(); // (3)
    }
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) { // (4)
        // 当出现异常就关闭连接
        cause.printStackTrace();
        ctx.close();
    }
}
```

DiscardServerHandler继承自ChannelInboundHandlerAdapter，这个类实现了ChannelInboundHandler接口，ChannelInboundHandler提供了许多事件处理的接口方法，我们可以覆盖这些方法。只需要继承ChannelInbound-HandlerAdapter类而不用自己去实现接口方法。

这里我们覆盖了chanelRead（）事件处理方法。每当从客户端收到新的数据时，这个方法会在收到消息时被调用，这个例子中，收到的消息类型是ByteBuf。

为了实现DISCARD协议，处理器不得不忽略所有接收到的消息。ByteBuf是一个引用计数对象，这个对象必须显式地调用release（）方法

来释放。注意处理器的职责是释放所有传递到处理器的引用计数对象。我们看看channelRead（）一般实现的方法，如代码清单13-2所示。

代码清单13-2　channelRead实现

```
@Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        try {
            // Do something with msg
        } finally {
            ReferenceCountUtil.release(msg);
        }
    }
```

在出现Throwable对象，即当Netty由于IO错误或者处理器在处理事件抛出异常时，exceptionCaught（）事件处理方法会被调用。在大部分情况下，捕获的异常应该被记录下来并且把关联的Channel关闭掉。通常在遇到不同的异常情况下会实现不同的处理方法，比如可能想在关闭连接之前发送一个错误码的响应消息。

目前为止我们已经实现了DISCARD服务器差不多一半的功能，接下来编写一个main（）方法来启动服务端的DiscardServerHandler，如代码清单13-3所示。

代码清单13-3　DiscardServer实现

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
/**
 * 丢弃任何进入的数据
 */
public class DiscardServer {
    private int port;
    public DiscardServer(int port) {
        this.port = port;
    }
    public void run() throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup(); // (1)
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap(); // (2)
            b.group(bossGroup, workerGroup)
             .channel(NioServerSocketChannel.class) // (3)
             .childHandler(new ChannelInitializer<SocketChannel>() { // (4)
```

```
                    @Override
                    public void initChannel(SocketChannel ch) throws Exception
                    {
                        ch.pipeline().addLast(new DiscardServerHandler());
                    }
                })
                .option(ChannelOption.SO_BACKLOG, 128)          // (5)
                .childOption(ChannelOption.SO_KEEPALIVE, true); // (6)
            // 绑定端口，开始接收进来的连接
            ChannelFuture f = b.bind(port).sync(); // (7)
            // 等待服务器 Socket 关闭。
            // 在这个例子中，这不会发生，但你可以优雅地关闭你的服务器。
            f.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
            bossGroup.shutdownGracefully();
        }
    }
    public static void main(String[] args) throws Exception {
        int port;
        if (args.length > 0) {
            port = Integer.parseInt(args[0]);
        } else {
            port = 8080;
        }
        new DiscardServer(port).run();
    }
}
```

NioEventLoopGroup是用来处理I/O操作的多线程事件循环器，Netty提供了许多不同的EventLoopGroup的实现来处理不同的传输类型。在这个例子中我们实现了一个服务端的应用，因此会有2个NioEventLoopGroup被使用。第一个经常被叫做“boss”，用来接收进来的连接；第二个经常被叫做“worker”，用来处理已经被接收的连接。一旦“boss”接收到连接，就会把连接信息注册到“worker”上。如何知道多少个线程已经被使用，如何映射到已经创建的Channel上都需要依赖于EventLoopGroup的实现，并且可以通过构造函数来配置他们的关系。

ServerBootstrap是一个启动NIO服务的辅助启动类。可以在这个服务中直接使用Channel，但处理过程比较复杂，一般不需要这样做。

代码中我们指定使用NioServerSocketChannel类来说明一个新的Channel如何接收进来的连接。

这里的事件处理类经常会被用来处理一个最近的已经接收的Channel。ChannelInitializer是一个特殊的处理类，他的目的是帮助使用者配置一个新的Channel。我们可以通过增加一些处理类比如DiscardServerHandler来配置一个新的Channel或者其对应的ChannelPipeline来实现网络程序。当网络程序变得复杂时，可以增加更

多的处理类到pipline上，然后提取这些匿名类到最顶层的类上。

可以设置代码中指定的Channel的配置参数，这是一个TCP/IP的服务端程序，因此我们要设置Socket的参数选项比如tcpNoDelay和keepAlive。详细内容可以参考ChannelOption和ChannelConfig实现的接口文档，来对ChannelOption有一个大致的认识。

option（）是提供给NioServerSocketChannel用来接收进来的连接。childOption（）是提供给由父管道ServerChannel接收到的连接，在这个例子中也就是NioServerSocketChannel。

剩下的就是绑定端口然后启动服务。这里是绑定了机器所有网卡上的8080端口。现在也可以多次调用bind（）方法来绑定不同的地址。

13.3.2　查看收到的数据

上一节我们已经编写了Discard服务端，现在需要测试一下它是否真的可以运行。最简单的测试方法是使用telnet命令。例如可以在命令行上输入telnet localhost 8080或者其他类型参数。但是我们不能确定这个服务端是否正常运行，因为它是一个Discard服务，没法得到任何响应。为了证明程序仍然在正常工作，我们需要修改服务端的程序来打印出它到底接收到了什么。

我们已经知道channelRead（）方法是在数据被接收的时候调用。让我们在DiscardServerHandler类的channelRead（）方法里添加一些代码，如代码清单13-4所示。

代码清单13-4　重新实现channelRead

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    ByteBuf in = (ByteBuf) msg;
    try {
        while (in.isReadable()) { // (1)
            System.out.print((char) in.readByte());
            System.out.flush();
        }
    } finally {
        ReferenceCountUtil.release(msg); // (2)
    }
}
```

这个低效的循环可以被简化为：System.out.println（in.toString（io.netty.util.CharsetUtil.US_ASCII））。

可以在这里调用in.release（），如果再次运行telnet命令，我们就能看到服务端会打印出它所接收到的消息。

13.4 RocketMQ基于Netty的通信功能实现

RocketMQ底层通信的实现是在Remoting模块里，因为借助了Netty，RocketMQ的通信部分没有很多的代码，就是用Netty实现了一个自定义协议的客户端/服务器程序。

13.4.1 顶层抽象类

RocketMQ的通信部分代码量并不多，代码结构如图13-2所示。

RocketMQ通信模块的顶层结构是RemotingServer和RemotingClient，分别对应通信的服务端和客户端。首先看看RemotingServer，如代码清单13-5所示。

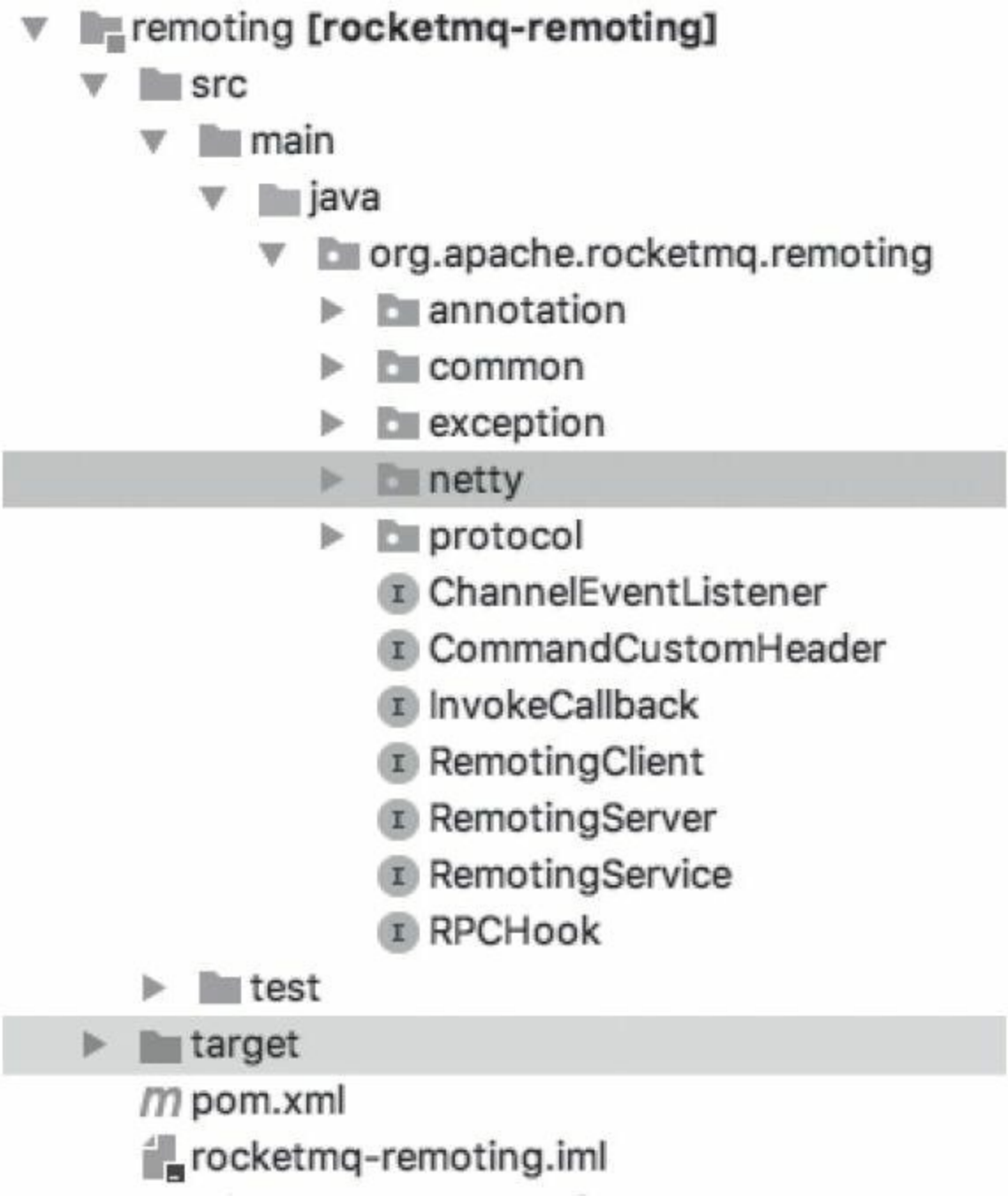


图13-2　Remoting模块代码结构

代码清单13-5　RemotingService类

```
public interface RemotingServer extends RemotingService {
    void registerProcessor(final int requestCode,
        final NettyRequestProcessor processor,
        final ExecutorService executor);
    void registerDefaultProcessor(final NettyRequestProcessor processor,
```

```
        final ExecutorService executor);
    int localListenPort();
    Pair<NettyRequestProcessor, ExecutorService> getProcessorPair(
        final int requestCode);
    RemotingCommand invokeSync(final Channel channel,
        final RemotingCommand request,
        final long timeoutMillis) throws InterruptedException,
        RemotingSendRequestException,
        RemotingTimeoutException;
    void invokeAsync(final Channel channel, final RemotingCommand request,
        final long timeoutMillis,
        final InvokeCallback invokeCallback) throws InterruptedException,
        RemotingTooMuchRequestException, RemotingTimeoutException,
        RemotingSendRequestException;

    void invokeOneway(final Channel channel, final RemotingCommand request,
        final long timeoutMillis)
        throws InterruptedException, RemotingTooMuchRequestException,
        RemotingTimeoutException,
        RemotingSendRequestException;
}
```

RemotingServer类中比较重要的是：localListenPort、registerProcessor和registerDefaultProcessor，registerDefaultProcessor用来设置接收到消息后的处理方法。

RemotingClient类和RemotingServer类相对应，比较重要的方法是updateNameServerAddressList、invokeSync和invokeOneway，updateName-ServerAddressList用来获取有效的NameServer地址，invokeSync与invokeOneway用来向Server端发送请求，如代码清单13-6所示。

代码清单13-6　RemotingClient类

```
public interface RemotingClient extends RemotingService {
    void updateNameServerAddressList(final List<String> addrs);
    List<String> getNameServerAddressList();
    RemotingCommand invokeSync(final String addr, final RemotingCommand request,
        final long timeoutMillis) throws InterruptedException,
        RemotingConnectException,
        RemotingSendRequestException, RemotingTimeoutException;
    void invokeAsync(final String addr, final RemotingCommand request,
        final long timeoutMillis,
        final InvokeCallback invokeCallback) throws InterruptedException,
        RemotingConnectException,
        RemotingTooMuchRequestException, RemotingTimeoutException,
        RemotingSendRequestException;
    void invokeOneway(final String addr, final RemotingCommand request,
        final long timeoutMillis)
        throws InterruptedException, RemotingConnectException,
        RemotingTooMuchRequestException,
        RemotingTimeoutException, RemotingSendRequestException;
    void registerProcessor(final int requestCode,
        final NettyRequestProcessor processor,
```

```
            final ExecutorService executor);
        void setCallbackExecutor(final ExecutorService callbackExecutor);
        boolean isChannelWritable(final String addr);
    }
```

13.4.2　自定义协议

NettyRemotingServer和NettyRemotingClient分别实现了RemotingServer与RemotingClient这两个接口，但它们有很多共有的内容，比如invokeSync、invokeOneway等，所以这些共有函数被提取到NettyRemotingAbstract共同继承的父类中。首先来分析一下在NettyRemotingAbstract中是如何处理接收到的内容的，如代码清单13-7所示。

代码清单13-7　处理请求消息

```
public void processRequestCommand(final ChannelHandlerContext ctx,
    final RemotingCommand cmd) {
    final Pair<NettyRequestProcessor, ExecutorService> matched = this
        .processorTable.get(cmd.getCode());
    final Pair<NettyRequestProcessor, ExecutorService> pair = null ==
        matched ? this.defaultRequestProcessor : matched;
final int opaque = cmd.getOpaque();
------
```

无论是服务端还是客户端都需要处理接收到的请求，处理方法由processRequestCommand定义，注意这里接收到的消息已经被转换成Remoting-Command了，而不是原始的字节流。

RemotingCommand是RocketMQ自定义的协议，具体格式如图13-3所示。

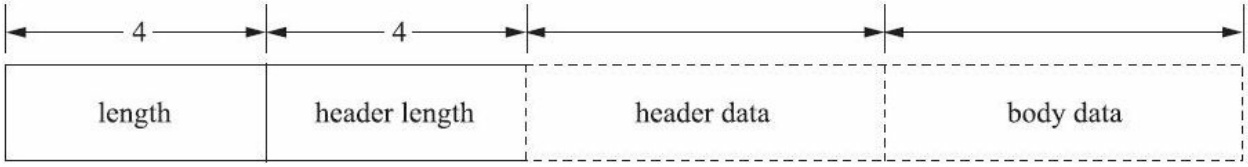


图13-3　RocketMQ自定义通信协议

这个协议只有四部分，但是覆盖了RocketMQ各个角色间几乎所有的通信过程，RemotingCommand有实际的数据类型和各部分对应，如代码清单13-8所示。

代码清单13-8　RemotingCommand成员变量

```
private int code;
private LanguageCode language = LanguageCode.JAVA;
private int version = 0;
private int opaque = requestId.getAndIncrement();
private int flag = 0;
private String remark;
private HashMap<String, String> extFields;
private transient CommandCustomHeader customHeader;
private SerializeType serializeTypeCurrentRPC = serializeTypeConfigInThis-Server;
private transient byte[] body;
```

RocketMQ各个组件间的通信需要频繁地在字节码和RemotingCommand间相互转换，也就是编码、解码过程，好在Netty提供了codec支持，这个频繁的操作只需要一行设置就可以完成：pipeline（）addLast（new NettyEncoder（），new NettyDecoder（）.……

下面分析一下发送消息的实现机制，即同部发送方式的实现，如代码清单13-9所示。

代码清单13-9　同步方式发送

```
public RemotingCommand invokeSyncImpl(final Channel channel,
    final RemotingCommand request,
    final long timeoutMillis)
    throws InterruptedException, RemotingSendRequestException,
    RemotingTimeoutException {
    final int opaque = request.getOpaque();
    try {
        final ResponseFuture responseFuture = new ResponseFuture(opaque,
            timeoutMillis, null, null);
        this.responseTable.put(opaque, responseFuture);
        final SocketAddress addr = channel.remoteAddress();
        channel.writeAndFlush(request).addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(
                ChannelFuture f) throws Exception {
                if (f.isSuccess()) {
                    responseFuture.setSendRequestOK(true);
                    return;
                } else {
                    responseFuture.setSendRequestOK(false);
                }
                responseTable.remove(opaque);
                responseFuture.setCause(f.cause());
                responseFuture.putResponse(null);
                log.warn("send a request command to channel <" + addr +
                    "> failed.");
            }
        });
        RemotingCommand responseCommand = responseFuture.waitResponse
            (timeoutMillis);
        if (null == responseCommand) {
            if (responseFuture.isSendRequestOK()) {
```

```
                    throw new RemotingTimeoutException(RemotingHelper
                        .parseSocketAddressAddr(addr), timeoutMillis,
                        responseFuture.getCause());
                } else {
                    throw new RemotingSendRequestException(RemotingHelper
                        .parseSocketAddressAddr(addr), responseFuture
                        .getCause());
                }
            }
            return responseCommand;
        } finally {
            this.responseTable.remove(opaque);
        }
    }
```

函数的RemotingCommand来自对要发送消息的封装，输入参数Channel来自io.netty.channel。Channel是通信的入口，Channel对象的获取，对于服务端和客户端来说差别很大。对客户端来说，由于是主动获取消息的一方，需要向哪个地址发送消息，于是通过Netty的Bootstrap方法创建一个连接（同时把连接后的Channel保存起来，避免每发一个消息都重新创建连接）；对服务端来说，很少主动发送消息，服务端一直在监听某个端口，当有一个连接请求进入后，服务端会把创建的Channel对象保存下来，供偶尔需要向客户端主动发消息的时候使用。

13.4.3　基于Netty的Server和Client

基于Netty实现的Server或Client程序，具体代码在NettyRemotingServer和NettyRemotingClient这两个类中，我们从ServerBootstrap的初始化来看RocketMQ是如何基于Netty实现Server端程序的，如代码清单13-10所示。

代码清单13-10　ServerBootstrap实现

```
ServerBootstrap childHandler =
    this.serverBootstrap.group(this.eventLoopGroupBoss, this
        .eventLoopGroupSelector)
        .channel(useEpoll() ? EpollServerSocketChannel.class :
            NioServerSocketChannel.class)
        .option(ChannelOption.SO_BACKLOG, 1024)
        .option(ChannelOption.SO_REUSEADDR, true)
        .option(ChannelOption.SO_KEEPALIVE, false)
        .childOption(ChannelOption.TCP_NODELAY, true)
        .childOption(ChannelOption.SO_SNDBUF, nettyServerConfig
            .getServerSocketSndBufSize())
        .childOption(ChannelOption.SO_RCVBUF, nettyServerConfig
            .getServerSocketRcvBufSize())
        .localAddress(new InetSocketAddress(this.nettyServerConfig
            .getListenPort()))
        .childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws Exception {
                ch.pipeline()
                    .addLast(defaultEventExecutorGroup,
                        HANDSHAKE_HANDLER_NAME,
                        new HandshakeHandler(TlsSystemConfig.tlsMode))
                    .addLast(defaultEventExecutorGroup,
                        new NettyEncoder(),
                        new NettyDecoder(),
                        new IdleStateHandler(0, 0, nettyServerConfig
                            .getServerChannelMaxIdleTimeSeconds()),
                        new NettyConnectManageHandler(),
                        new NettyServerHandler()
                    );
            }
        });
```

ServerBootStrap的BossEventLoop使用的是单线程的NioEventLoopGroup，workerEventLoop在Linux平台使用的是默认3个线程的EpollEventLoopGroup，在非Linux平台使用的是3个线程的NioEventLoopGroup。在最后几行代码中还可以看到添加了NettyEncoder和NettyDecoder这两个Handler。这些Handler执行在一个8线程的DefaultEventExecutorGroup中。

RocketMQ对通信过程的另一个抽象是Processor和Executor，当接收到一个消息后，直接根据消息的类型调用对应的Processor和Executor，把通信过程和业务逻辑分离开来。我们通过一个Broker中的代码段来看看注册Processor的过程，如代码清单13-11所示。

代码清单13-11　注册Processor

```
public void registerProcessor() {
    /**
     * SendMessageProcessor
     */
    SendMessageProcessor sendProcessor = new SendMessageProcessor(this);
    sendProcessor.registerSendMessageHook(sendMessageHookList);
    sendProcessor.registerConsumeMessageHook(consumeMessageHookList);

    this.remotingServer.registerProcessor(RequestCode.SEND_MESSAGE,
        sendProcessor, this.sendMessageExecutor);
    this.remotingServer.registerProcessor(RequestCode.SEND_MESSAGE_V2,
        sendProcessor, this.sendMessageExecutor);
    this.remotingServer.registerProcessor(RequestCode.SEND_BATCH_MESSAGE,
        sendProcessor, this.sendMessageExecutor);
    this.remotingServer.registerProcessor(RequestCode
        .CONSUMER_SEND_MSG_BACK, sendProcessor, this
        .sendMessageExecutor);
```

注册Processor示例代码段来自org.apache.rocketmq.broker包中的BrokerController类，可以看出通过RocketMQ所做的抽象、通信逻辑和信息处理逻辑被分离开，使结构变得非常清晰。

13.5 本章小结

本章介绍了RocketMQ底层通信的实现机制，由于它是基于Netty来实现的，所以首先介绍了Netty的基础知识。Netty被用在很多开源软件的底层通信部分，RocketMQ以Netty为基础，还实现了一种机制，把通信功能和消息处理功能分离，不同类型的通信内容被抽象成发送带有对应类型代码的Command，同时根据类型代码查找对应的Processor和Executor来执行，结构非常清晰，为我们自己实现网络通信程序提供了参考。